UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

CiTiUS Centro Singular de Investigación en Tecnoloxías da Información

PhD Dissertation

HARDWARE COUNTER BASED PERFORMANCE ANALYSIS, MODELLING, AND IMPROVEMENT THROUGH THREAD MIGRATION IN NUMA SYSTEMS

Author:

Oscar García Lorenzo

PhD Supervisors:

José Carlos Cabaleiro Domínguez

Tomás Fernández Pena

January 2016

José Carlos Cabaleiro Domínguez, Profesor Titular de Universidad del Área de Arquitectura de Computadores de la Universidad de Santiago de Compostela

Tomás Fernández Pena, Profesor Titular de Universidad del Área de Arquitectura de Computadores de la Universidad de Santiago de Compostela

HACEN CONSTAR:

Que la memoria titulada **HARDWARE COUNTER BASED PERFORMANCE ANALYSIS, MODELLING, AND IMPROVEMENT THROUGH THREAD MIGRATION IN NUMA SYSTEMS** ha sido realizada por **D. Oscar García Lorenzo** bajo nuestra dirección en el Centro Singular de Investigación en Tecnoloxías da Información de la Universidad de Santiago de Compostela, y constituye la Tesis que presenta para obtar al título de Doctor.

January 2016

HEREBY CERTIFY:

That the dissertation entitled **HARDWARE COUNTER BASED PERFORMANCE ANALYSIS, MODELLING, AND IMPROVEMENT THROUGH THREAD MIGRATION IN NUMA SYSTEMS** has been developed **D. Oscar García Lorenzo** under our direction at the Centro Singular de Investigación en Tecnoloxías da Información (CiTiUS) of the Universidad de Santiago de Compostela, in fullfilment of the requirements for the Degree of Doctor of Philosophy.

January 2016

José Carlos Cabaleiro Domínguez
Codirector de la tesis

Tomás Fernández Pena
Codirector de la tesis

Oscar García Lorenzo
Autor de la tesis

Agradecimentos/Acknowledgments

Galician Supercomputing Centre, Centro Tecnolóxico de Supercomputación de Galicia (CESGA).
Red de Computación de Altas Prestaciones sobre Arquitecturas Paralelas Heterogéneas (CAPAP-H). (CAPAP-H4 TIN2011-15734-E).
Galician network under the Consolidation Program of Competitive Research Units (Network ref. R2014/041)
European network HiPEAC-2
Computer Architecture & Operating Systems Department (CAOS), Departamento de Arquitectura de Computadores y Sistemas Operativos, Universitat Autònoma de Barcelona.
High Performance and Distributed Computing Research Cluster, Queen's University of Belfast
This work has been partially supported by the Ministry of Education and Science of Spain, FEDER funds under contracts TIN 2010-17541 and TIN 2013-41129P, and by the Xunta de Galicia (Spain) under contracts 2010/28, EM2013/041 and GRC2014/008, and project 09TIC002CT

January 2016

Contents

*

Introduction

While for most of the history of computing programming and execution was sequential, one instruction followed another, with few exceptions, on the last ten years this paradigm has completely changed. Modern computer systems are based on multicore processors, which means parallel programming and execution is now dominant, when before it was mostly the realm of high performance computing. This change has brought many challenges, not all of them solved. Parallel programming is inherently more difficult than sequential programming. If, during the 20th century, it could be taken for granted that someone who needed parallel programming would at least have access to expert knowledge, this is no longer the case. As such, all approaches to make parallel programming more accessible are welcomed.

A parallel computer system where all its components are equal and have the same performance is simpler to program for. Unfortunately, these systems do not allow for the higher peak performance or for enough flexibility to carry out a variety of tasks. This is why, nowadays, many computers are of a heterogeneous nature, mixing different architectural approaches in the same system. But even on those computers apparently simpler, like shared memory systems with multiple processors, imbalances negatively affect performance. These systems are prevalent on internet servers and workstations, and are the foundation of high performance supercomputers. In this work, a series of tools, applications and models designed to help the programming of these systems, and even to improve their performance without direct user intervention, are presented.

The advantages modern processors give for performance monitoring allow the users to gain insight on the execution of their applications. Nevertheless, the performance information processors give may not be used to analyse program improvements in a straightforward way. This information may be complex and, by virtue of its detail, extensive. During this work, the tools and models presented take advantage of these facilities to offer the users a clear view of the behaviour of their codes to tackle actual issues that affect performance. With the experience acquired developing these tools and models, an application to automatically improve the performance of parallel applications or mixed workloads was implemented and tested.

In Chapter 1, the shared memory computers that are the scope of this work are presented, along with the performance monitoring facilities of modern Intel processors. Chapter 2 presents a set of memory accesses analysis tools, designed to allow its users to understand the data locality and data placement of their codes. Its usefulness in a series of cases is shown. A new performance model, based on the Berkeley Roofline Model, is introduced in Chapter 3, alongside with a set of tools to simplify the task of obtaining it, is presented. A series of applications of the model are presented to highlight its usefulness. Finally, in Chapter 4, a tool to improve the performance in parallel computers is presented. This tool automatically places and migrates threads during execution, using different strategies. Those proposals are detailed and tested, clearly showing the importance of thread and data placement for performance.

Chapter 1

Multicore Processors, NUMA Systems and Hardware Counters

Throughout this work we will be dealing with shared memory computer systems composed of various multicore processors. This chapter describes the main characteristics of these systems, as well as the configuration of the actual computers used. More specifically, the performance monitoring facilities of the modern Intel processors are described. These facilities allow to obtain during runtime a number of useful performance information, which can be used to increase the performance the execution of programs.

1.1 Multicore processors and NUMA systems

This work focuses on multiprocessors, which has been defined [22] as computers consisting of tightly coupled processors whose coordination and usage are typically controlled by a single operating system and that share memory through a shared address space. Such systems exploit thread-level parallelism through two different software models. The first one is the execution of a tightly coupled set of threads collaborating on a single task, which is typically called parallel processing. The second one is the execution of multiple, relatively independent processes that may originate from one or more users, which is a kind of request-level parallelism. Request-level parallelism may be exploited by a single application running on multiple processors, such as a database responding to queries, or multiple applications running independently, often called multiprogramming. The multiprocessors typically range in size from a dual processor to dozens of processors with several cores each, and communicating and coordinating through the sharing of memory. Although sharing through memory implies a shared address space, it does not necessarily mean that there is a single physical memory. Such multiprocessors include both single-chip systems with multiple cores, known as multicore, and computers consisting of multiple chips, each of which may be a multicore processor. In addition to true multiprocessors, the multithreading technique supports multiple threads executing in an interleaved way on a single multiple issue processor. Many multicore processors also include support for multithreading.

To take advantage of a multiprocessor with n cores, we must usually have at least n threads or processes to execute [22]. The independent threads within a single process are typically identified by the programmer or created by the operating system (from multiple independent requests). At the other extreme, a thread may consist of a few tens of iterations of a loop, generated by a parallel compiler exploiting data parallelism in the loop. Although the amount of computation assigned to a thread, called the grain size, is important in considering how to exploit thread-level parallelism efficiently, the essential qualitative distinction from instruction-level parallelism is that thread-level parallelism is identified at a high level by the software system or programmer. Also, each thread consists of hundreds to millions of instructions that may be executed in parallel. Threads can also be used to exploit data-level parallelism, although the overhead is likely to be higher than would be seen with an Single Instruction, Multiple Data (SIMD) [22] processor or with a GPU.

Existing shared-memory multiprocessors fall into two classes, depending on the number of cores involved, which in turn dictates a memory organisation and interconnect strategy. We refer to the multiprocessors by their memory organisation because what constitutes a small or large number of processors is likely to change over time. These two classes are called symmetric (shared-memory) multiprocessors (SMPs) [22] and distributed shared memory (DSM) [22]. In both architectures, communication among threads can be performed through a shared address space, meaning that a memory reference can be made by any processor to

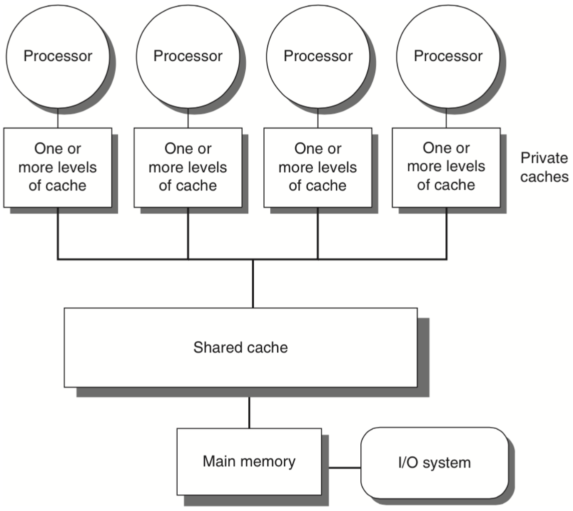

Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Private caches
Shared cache
Main memory
I/O system

Figure 1.1: Architecture of a centralised shared-memory multiprocessor based on a multicore chip [22].

any memory location. The term shared memory associated with both SMP and NUMA refers to the fact that the address space is shared. These two architectures are detailed in the next subsections.

1.1.1 Symmetric multiprocessors

The first group, which has been called symmetric (shared-memory) multiprocessors (SMPs) [22], or centralised shared-memory multiprocessors, features a small numbers of processors or cores, typically eight or fewer. Those processors are highly coupled. For multiprocessors with such small processor counts, it is possible to share a single centralised physical memory that all processors have equal access to, hence the term symmetric. In multicore chips, the memory is effectively shared in a centralised fashion among the cores, and all existing multicores are SMPs. An example of SMPs are the multicore chips (Figure 1.1), where the memory is effectively shared in a centralised fashion among the cores.

SMP architectures are also sometimes called uniform memory access (UMA) multiprocessors, arising from the fact that all processors have a uniform latency from memory, even if the memory is organised into multiple banks.

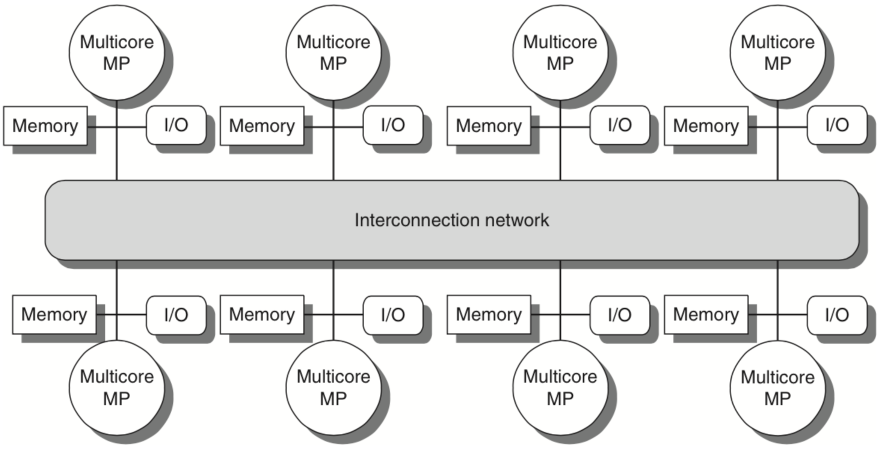


Figure 1.2: The basic architecture of a distributed-memory multiprocessor typically consists of a multicore multiprocessor chip with memory and possibly I/O attached and an interface to an interconnection network that connects all the nodes [22].

1.1.2 Distributed shared memory

The alternative design approach consists of multiprocessors with physically distributed, and logically shared, memory, and it has been called distributed shared memory (DSM) [22]. Processors are loosely coupled. Figure 1.2 shows the basic organisation of these multiprocessors.

To support larger processor counts, memory must be distributed among the processors rather than centralised; otherwise, the memory system would not be able to support the bandwidth demands of a larger number of processors without incurring in excessively long access latency. Given the fast increase in processor performance and the associated increase in a processor's memory bandwidth requirements, the size of a multiprocessor for which distributed memory is preferred continues to shrink. The introduction of multicore processors has meant that even two-chip multiprocessors may use distributed memory. The larger number of processors or cores also raises the need for a high- bandwidth interconnect. Both directed networks (i.e., switches) and indirect networks (often multidimensional meshes) are used.

Distributing the memory among the nodes both increases the bandwidth and reduces the latency to local memory. A DSM multiprocessor is also called a NUMA (NonUniform Memory Access), since the access time depends on the location of a data word in memory. This will be the preferred term in this work, because it will be dealing mostly with issues related to memory accesses. The key disadvantages for a NUMA system are, first, that communicating data among processors becomes somewhat more complex, and, second, that a NUMA system requires more effort in the software to take advantage of the increased memory bandwidth afforded by distributed memories. Under NUMA, a processor can access its own local memory faster than non-local memory (local to another processor). Because almost all multicore-based multiprocessors with more than one processor chip (or socket) use distributed memory, we will explain the operation of distributed memory multi- processors from this viewpoint.

With NUMA, maintaining cache coherence across shared memory has a significant overhead. Cache coherent NUMA (ccNUMA) systems use inter-processor communication among cache controllers to keep a consistent memory image when more than one cache stores the same memory location. For this reason, ccNUMA may perform poorly when multiple processors attempt to access the same memory area in rapid succession. Support for NUMA in operating systems attempts to reduce the frequency of this kind of accesses by allocating processors and memory in concurrent ways, improving the performance by means of exploiting the so called affinity [54].

Many OS have included support for ccNUMA, specifically in Linux, processor cores and memory are logically distributed in nodes, which closely mimic the hardware distribution. In multiprocessor systems there is usually one multicore processor per node, that consists of the cores in one processor and their local memory. As with ccNUMA processes, threads and data may be placed in any node or combination of them, it is expected that data accesses between cores and memory of the same node to be faster. These node assignment can be modified by an user to more closely reflect the hardware (by creating more nodes or changing the distribution of cores), and they can be characterised by distance metrics, that represent how, in terms of delay, far is the memory location from one node to the others.

1.1.3 Memory gap

The first main memories to be used on digital computers were constructed using a technology much slower than that used for the logic circuits, and it was taken for granted that there would be a memory gap [84]. Mercury delay line memories spent a lot of their time waiting for the required word to come round and were very slow indeed. CRT (Williams Tube) memories and the core memories that followed them were much better. By the early 1970s semiconductor memories were introduced in the design. This did not result in memory performance catching up fully with processor performance, although in the 1970s it came close. It might have expected that from that point memories and processors would scale together, but this was not the case. The reason for that is the significant differences in the DRAM semiconductor technology used for memories compared with the technology used for microprocessor circuits. The memory gap makes itself felt when a cache miss occurs and the missing word must be supplied from main memory. It thus affects users whose programs do not fit into the last cache level. As far as a workstation user is concerned, the most noticeable effect of an increased memory gap is to make the observed performance more dependent on the application area than it would be otherwise.

Since 1980, the memory gap has been increasing steadily, as the speed of processors increase. On the other hand, shrinkage enables L2 caches to increase in size and, to some extent, this balances out the effect of the increased memory gap. However, there are reasons that indicate that non-cacheable problems are increasing in importance [84]. For example, large simulations in science and technology, big data computing, multimedia, etc. Also many database servers used in transaction processing rarely, if ever, manage to establish a working set that will fit entirely within a cache. It must be accepted that, however large the cache memory may be, there will be plenty of problems for which it may never be enough. For such problems the cache actually gets in the way and slows down the running of the program.

1.1.4 Locality and affinity

To take advantage of parallelism and of the memory hierarchy, one of the most important code property regularly exploited is the locality: codes tend to reuse data and instructions they have used recently [22]. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. An implication of locality is that it can be predicted with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past. The principle of locality also applies to data accesses, though not as strongly as to code accesses. Two types of locality can be considered:

- Temporal locality states that recently accessed items are likely to be accessed in the near future.
- Spatial locality says that items whose addresses are near one another tend to be referenced close together in time.

Modern OS enable the binding and unbinding of a process or a thread to a specific central processing unit (CPU), core or a range of cores, so that the process or thread will execute only on the designated core or cores rather than in any others. This is done by defining for a thread an affinity to a particular set of CPUs or cores. This mechanism helps to make the most of the principle of locality.

1.1.5 Thread migration

In NUMA systems, taking into account architectural features, particularly the behaviour of memory accesses, it is critical to improve locality among accesses and affinity between data and cores to improve performance. In particular, both locality and affinity are important to reduce the access latency to data. In addition, a large fraction of the on-chip multicore interconnect traffic is originated not from actual data transfers but from communication among cores to maintain data coherence [75]. An important impact of this overhead is the on-chip interconnect power and energy consumption.

Moving threads close to where their data reside can help to alleviate those issues. When threads migrate, the corresponding data usually stays in the original memory module, and is accessed remotely by the migrated thread. This could be a source of inefficiencies that, sometimes, cannot be overlapped by the benefits of the migration [11, 78, 79, 38, 33]. Also, proposals for heterogeneous multicore move threads between cores to exploit power-performance-area trade-offs [34]. Performance information can be used to guide thread migration strategies to improve the efficiency of the execution of the code by increasing data locality and/or thread affinity. Each such migration incurs overhead, similar to a context switch [37, 39], from saving and restoring processor states and virtual machine control structures, extra translation lookaside buffer misses and related page walks, cache misses, and interrupt rerouting.

The indirect overhead of TLB and cache misses produced by each migration is potentially higher than for a context switch, because the migrated thread begins execution in a different processor environment and cache hierarchy. The performance benefit of saving and restoring cached data during migration is analyzed in [74]. Thread startup performance can be accelerated after migration by predicting and prefetching the working set of the application into the new cache [6].

1.2 Intel processors

In this work, we have focused on Intel processors, more specifically on the Itanium 2 [25] and the Sandy Bridge [28] architectures. In these processor families, cores are equipped with a Performance Monitoring Unit (PMU) that allows a user to obtain various metrics to evaluate the behaviour of the processor. In particular, these Intel processors allow to monitor the memory performance of the processor at core level, including memory latency, which makes them useful to study the behaviour in NUMA systems. In next subsections, the particular systems used through this work are presented and detailed.

1.2.1 Itanium

Itanium is a family of 64-bit Intel microprocessors that implement the Intel Itanium architecture (formerly called IA-64). IA-64 implements a form of Very Long Instruction Word (VLIW) architecture, instead of the Reduced Instruction Set Computing (RISC) architectures of x86. Intel marketed the processors for enterprise servers and high-performance computing systems. The Itanium architecture originated at Hewlett-Packard (HP), and was later jointly developed by HP and Intel. In November 2007, the Itanium 2 9100 series, codenamed Montvale, was released. This is the one we are using in this work.

The Itanium 2 Montvale processor has two cores, each with a three level cache memory hierarchy [25]. In each core the first level consists of two 16 KB caches, one for data (L1D), and another for instructions (L1I). The second level has a 256 KB cache dedicated exclusively to data (L2D), and another 1 MB for instructions (L2I). The size of the third level cache (L3) varies among each processor family between 1.5 MB and 24 MB (see Figure 1.3). Minimum

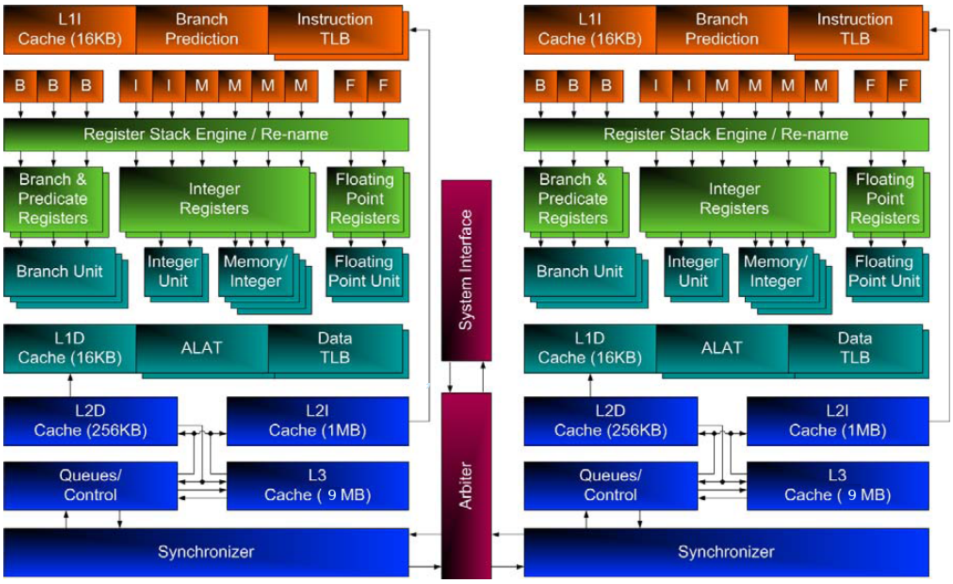


Figure 1.3: The Itanium 2 Montvale processor memory hierarchy.

latency for a cache miss in the L1D is 5 cycles, reached when the data are on the L2D. Meanwhile, if the data are on the L3, the latency is, at least, 14 cycles. Floating point data are stored directly in the L2D, so their access always implies a L1D cache miss. All the case studies in this work for the Itanium architecture were executed on a HP Integrity rx7640 [23, 15] computation node, with 8 Itanium 2 Montvale processors (16 cores) and 128 GB of RAM. The HP Integrity rx7640 consists of two cells, each with 64 GB of memory and 4 dual-core processors (8 cores per cell), connected by a buffer. Cache coherence is maintained by a mixed protocol, using a memory directory between cells and a snooping protocol inside each cell. Cores 0 to 7, that are in node 0, have greater affinity to one half of the main memory (local to 0 to 7) and cores 8 to 15, that are in node 1, to the other half (remote to 0 to 7). Furthermore, the different delays of access to local and remote main memories mean that the data access time is one of the aspects that influences the performance of shared memory parallel codes the most, specially in those with irregular accesses. This system will be called "our Itanium Server" for the remainder of this work.

1.2.2 Sandy Bridge

Sandy Bridge [28] is the codename for a microarchitecture developed by Intel beginning in 2005 for central processing units in computers to replace the Nehalem microarchitecture. It uses the x86 instruction set. Intel demonstrated a Sandy Bridge processor in 2009, and released first products based on the architecture in January 2011. Developed primarily by the Israeli branch of Intel, the codename was originally “Gesher” (meaning “bridge” in Hebrew).

The Sandy Bridge cache is divided into 3 levels, with the L1 (32 KB for data) and L2 (256 KB) private to each core, and the L3 shared among the cores in a processor, up to 8 cores. The cache line size is 64 bytes, storing 16 floating point numbers per line. In the L1 there is a line buffer where the last cache line read is stored before being written to cache. This way, if an outstanding core cache miss to same cache-line address was already underway, data can be read directly from the buffer, reducing latency. Furthermore, the L2 can prefetch the next cache line, thus reading two cache lines (128 bytes) in just one memory transaction.

The case studies in this work for the Xeon Sandy Bridge architecture were carried out in three different systems. For future reference, they are:

- Xeon Server X - A server with two quad-core Xeon E5-2603 (8 cores in total, Sandy Bridge architecture, 10 MB L3 cache, 1.8 GHz) and 16 GB of RAM. It has two nodes, each local to 8 GB of RAM, one consists of cores 0 to 3 (node 0), and the other of cores 4 to 7 (node 1).

- Xeon Server Y - A server with two octo-core Xeon E5-2650 (16 physical cores in total, 32 considering hyperthreading, Sandy Bridge architecture, 20 MB L3 cache, 2 GHz - 2.8 GHz) and 64 GB of RAM. It has two nodes, each comprising 32 GB of RAM. Node 0 is made of the eight even indexed cores while node 1 is made of the eight odd indexed cores.

- Xeon Server Z - A server with four octo-core Xeon E5-4620 (32 physical cores in total, 64 when hyperthreading is used, Sandy Bridge architecture, 16 MB L3 cache, 2.2 GHz-2.6 GHz) and 512 GB of RAM. It has four nodes, each local to 128 GB of RAM. Node 0 has cores 0 to 7, node 1 has cores 8 to 15, node 2 has cores 16 to 23 and node 3 has cores 24 to 31.

Processors on Xeon Servers Y and Z use frequency scaling, so the frequency of the cores may change in a range of values. Intel uses a technology, called Turbo Boost, which dynamically increases the processor's frequency as needed by taking advantage of thermal and power headroom. This means that, when only a few number of cores are used and the processor does not dissipate too much heat, the CPU frequency increases. Nevertheless, if all the cores are in use, the frequency is set to its lowest value. This may influence performance and must be taken into account when measured.

1.3 Hardware counters

Hardware performance counters, or hardware counters (HC) for short, are special-purpose registers built into modern microprocessors to store information of hardware-related activities within computer systems. The number of available hardware counters in a processor is limited while each CPU model might have a lot of different events that a developer might like to measure. Each counter can be programmed with an event type to be monitored, like L1 cache misses or branch mispredictions, for example.

Compared to software profilers, hardware counters provide low-overhead access to a wealth of detailed performance information related to CPU's functional units, caches, main memory, etc. Another benefit of using them is that no source code modifications are needed in general. However, the types and meanings of hardware counters vary from one architecture to another due to the variation in hardware organisations. Also, the limited number of registers to store the counters often force users to conduct multiple measurements to collect all desired performance metrics.

Due that modern superscalar processors schedule and execute multiple instructions out-of-order at one time there can be difficulties correlating the low level performance metrics back to source code. These "in-flight" instructions can be retired at any time, depending on memory accesses, hits in cache, stalls in the pipeline and many other factors. This can cause performance counter events to be attributed to the wrong instructions, making precise performance analysis difficult or even impossible.

In the Itanium 2 two sets of performance monitor registers are defined [25]. Performance

Monitor Configuration (PMC) registers are used to configure the monitors. Performance Monitor Data (PMD) registers provide data values from the monitors. The Itanium 2 processor provides four generic performance counters (PMC/PMD pairs), and the following model-specific monitoring registers (all present on the Itanium 2 Montvale): instruction and data event address registers (EARs) for monitoring cache and TLB misses, a branch trace buffer, two opcode match registers, and an instruction address range check register.

The counter width on the Itanium 2 processor is 48 bits (bit 47 indicates overflow condition). PMC/PMD pairs on the Itanium 2 processor are symmetrical, i.e., nearly all event types can be monitored by all counters. The main task of PMCs is to select the events to be monitored by the respective PMDs. Only the use of data event address registers (EARs) will be detailed in this document (in Subsection 1.3.1).

In more modern Intel processors, hardware counters are called model-specific registers (MSRs) [28]. For performance monitoring they add some additional nomenclature. The MSRs are registers available primarily to operating system or executive procedures (that is, code running at privilege level 0). These registers control items such as the debug extensions, the performance-monitoring counters, the machine-check architecture, and the memory type ranges (MTRRs). The number and function of these registers varies among different members of the Intel 64 and IA-32 processor families. Most systems restrict access to system registers by application programs. Systems can be designed, however, where all programs and procedures run at the most privileged level (privilege level 0). In such a case, application programs would be allowed to modify the system registers.

Among the MSRs are the performance event select registers, which allow configuring a performance monitoring event. There is a finite number of performance event select MSRs (IA32_PERFEVTSELx MSRs), only 4 on the Sandy Bridge architecture. This means that, at most 4 events can be measured at the same time, as long as there are no conflicts among them that preclude their concurrent counting. The result of a performance monitoring event is reported in a performance monitoring counter (IA32_PMCx MSR). Performance monitoring counters are paired with performance monitoring select registers. The precise event based sampling (PEBS) facilities for Sandy Bridge processors will be detailed in Subsection 1.3.2.

1.3.1 EAR counters

The Itanium 2 provides a set of Event Address Registers (EARs) that record the instruction and data addresses of data cache misses for loads, the instruction and data addresses of data TLB misses, and the instruction addresses of instruction TLB and cache misses [25]. When used to capture cache misses, EARs allow latency detection from 4 cycles upwards, so using them any miss, or floating point access (floating point data are always stored on the L2D), can be potentially detected. Table 1.1 summarises the capabilities offered by the Itanium 2 processor EARs and the branch trace buffer. Exposing miss event addresses to software allows them to be monitored either by sampling or by code instrumentation. This eliminates the need for trace generation to identify and solve performance issues and enables performance analysis by a much larger audience.

The Itanium 2 processor EARs enable statistical sampling by configuring a performance counter to count the occurrences of a given event. The performance counter value is set up to interrupt the processor after a predetermined number of events have been observed. The data cache event address register repeatedly captures the instruction and data addresses of actual data cache load misses. Whenever the counter overflows, miss event address collection is suspended until the event address register is read by software (this prevents software from capturing a miss event that might be caused by the monitoring software itself). So, when the counter overflows, an interruption is delivered, the observed event addresses are collected, and a new observation interval can be setup by rewriting the performance counter register. For time-based (rather than event-based) sampling methods, the event address registers indicate whether or not a qualified event was captured. Statistical sampling can achieve arbitrary event resolution by varying the number of events within an observation interval and by increasing the number of observation intervals.

On the Itanium 2, performance monitoring can be confined to a subset of events. Events can be qualified for monitoring based on an instruction address range, a particular instruction opcode, a data address range, an event-specific "unit mask" (umask), the privilege level and instruction set the event was caused by, and the status of the performance monitoring freeze bit (PMC0.fr). In particular, the Itanium 2 processor allows event collection for memory operations to be constrained to a programmable data address range. This enables selective

Table 1.1: Itanium 2 Processor EARs and Branch Trace Buffer.

Event Address Register	Triggers On	What is Recorded
Instruction Cache	Instruction fetches that miss the L1 instruction cache (demand fetches only)	Instruction Address, Number of cycles fetch was in flight
Instruction TLB (ITLB)	Instruction fetch missed L1 ITLB (demand fetches only)	Instruction Address Who serviced L1, ITLB miss: L2 ITLB VHPT or software
Data Cache	Load instructions that miss L1 data cache	Instruction Address, Data Address, Number of cycles load was in flight.
Data TLB (DTLB)	Data references that miss L1 DTLB	Instruction Address, Data Address, Who serviced L1 DTLB miss: L2 DTLB, VHPT or software.
Branch Trace Buffer	Branch Outcomes	Branch Instruction Address, Branch Target Instruction Address, Mispredict status and reason

monitoring of data cache miss behaviour of specific data structures. Four architectural Data Breakpoint Registers (DBRs) can be used to specify the desired address range. Data address range checking capability is controlled by a Memory Pipeline Event Constraints Register (PMC13). When enabled (using bits 1 and 0, to indicate one of the four DBRs to be used), data address range checking is applied to loads, stores, semaphore operations, and the *lfetch* instruction.

To program the PMU and gather the execution results of the target codes, the `libpfm2` library and the `perfmon2` [14] communication interface were used. `libpfm2` is a helper library that can be used to implement CPU monitoring tools. This library contains all the information about the specific PMUs in each processor model. The programmer only needs to indicate what to measure, using generic events common to most processor models, and the library maps these events to the ones implemented in the actual PMU. The `libpfm2` library supports many processor families, like IA-64, X86-64, P6, and even others from AMD, ARM, etc. The `perfmon2` interface is a performance monitoring subsystem for Linux used to get access to the PMU counters and registers, to program and read them. `Perfmon2` uses the system specific information provided by `libpfm2` to program counters.

1.3.2 PEBS

Intel processors based on Intel Core microarchitecture support precise event based sampling (PEBS [28]). This feature was introduced in processors based on Intel NetBurst microarchitecture. PEBS uses a debug store mechanism and a performance monitoring interruption to store a set of architectural state information for the processor. This information provides the architectural state of the instruction executed after the instruction that caused the event. On Intel Sandy Bridge, all general-purpose performance counters can be used for PEBS if the performance event is supported. The MSR IA32_PEBS_ENABLE provides 4 bits that must be used to enable which overflow condition will cause the PEBS record to be captured. Additionally, the MSR IA32_PEBS_ENABLE provides 4 additional bits that software must use to enable latency data recording in the PEBS record upon the respective IA32_PMCx overflow condition. The layout of IA32_PEBS_ENABLE for processors based on Intel Nehalem or Sandy Bridge is shown in Figure 1.4. When a counter is enabled to capture machine state (PEBS_EN_PMCx = 1), the processor will write machine state information to a memory buffer specified by software as detailed below. When the counter IA32_PMCx overflows from maximum count to zero, the PEBS hardware is armed.

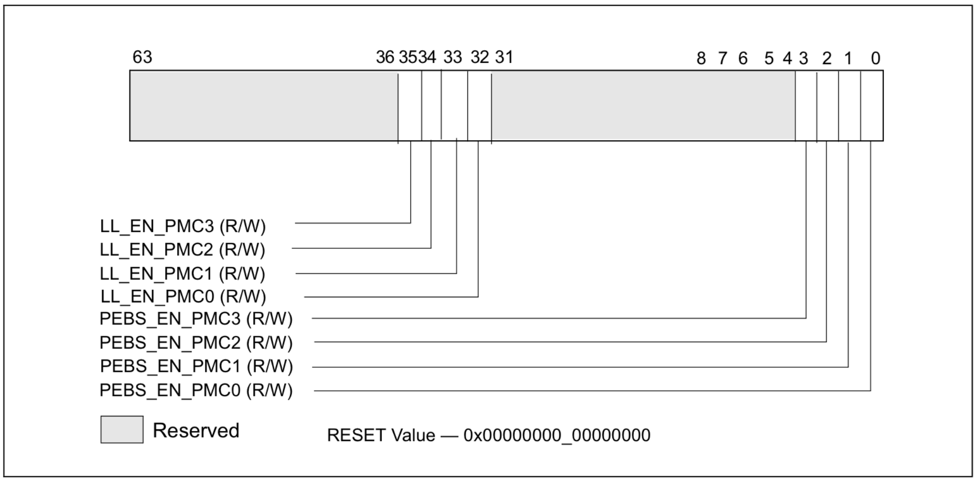


Figure 1.4: Layout of IA32_PEBS_ENABLE MSR.

Upon occurrence of the next PEBS event, the PEBS hardware triggers an assist, causing a PEBS record to be written. The return instruction pointer (RIP) reported in the PEBS record will point to the instruction after the instruction that causes the PEBS assist. The machine state reported in the PEBS record is the machine state after the instruction that causes the PEBS assist is retired. For instance, if the instructions

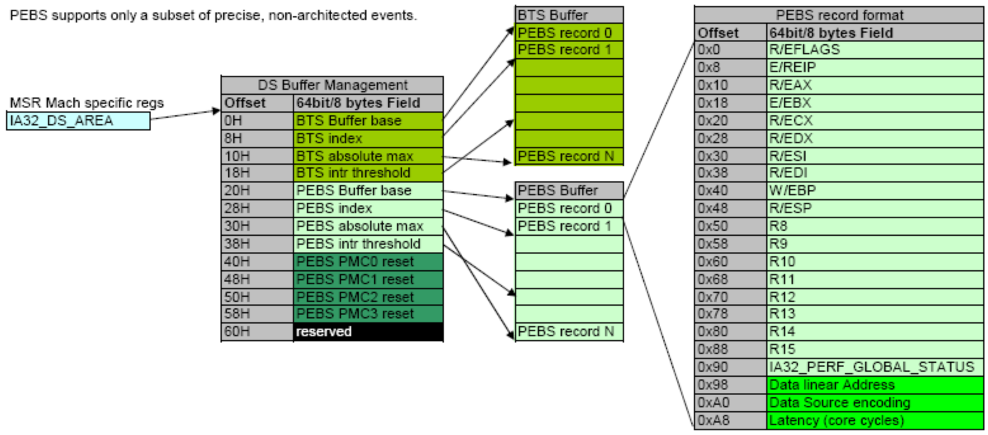
```
mov eax, [eax] ; causes PEBS assist
nop
```

are executed, the PEBS record will report the address of the `nop`, and the value of EAX in the PEBS record will show the value read from memory, not the target address of the read operation. The PEBS record format is shown in Figure 1.5 and in Table 1.2. Each field in the PEBS record is 64 bits long. The PEBS record format, along with debug/store area storage format, does not change regardless of IA-32e mode is active or not.

In IA-32e mode, the full 64-bit value is written to the register. If the processor is not operating in IA-32e mode, 32-bit value is written to registers with bits 63:32 zeroed. Registers

Table 1.2: PEBS Record Format for Intel Core i7 Processor Family.

Byte Offset	Field	Byte Offset	Field
0x0	R/EFLAGS	0x58	R9
0x8	R/EIP	0x60	R10
0x10	R/EAX	0x68	R11
0x18	R/EBX	0x70	R12
0x20	R/ECX	0x78	R13
0x28	R/EDX	0x80	R14
0x30	R/ESI	0x88	R15
0x38	R/EDI	0x90	IA32_PERF_GLOBAL_STATUS
0x40	R/EBP	0x98	Data Linear Address
0x48	R/ESP	0xA0	Data Source Encoding
0x50	R8	0xA8	Latency value (core cycles)


PEBS supports only a subset of precise, non-architected events.
MSR Mach specific regs
IA32_DS_AREA

DS Buffer Management	
Offset	64bit/8 bytes Field
0H	BTS Buffer base
8H	BTS index
10H	BTS absolute max
18H	BTS intr threshold
20H	PEBS Buffer base
28H	PEBS index
30H	PEBS absolute max
38H	PEBS intr threshold
40H	PEBS PMC0 reset
48H	PEBS PMC1 reset
50H	PEBS PMC2 reset
58H	PEBS PMC3 reset
60H	reserved

BTS Buffer
PEBS record 0
PEBS record 1
PEBS record N

PEBS Buffer
PEBS record 0
PEBS record 1
PEBS record N

PEBS record format	
Offset	64bit/8 bytes Field
0x0	R/EFLAGS
0x8	E/REIP
0x10	R/EAX
0x18	E/EBX
0x20	R/ECX
0x28	R/EDX
0x30	R/ESI
0x38	R/EDI
0x40	W/EBP
0x48	R/ESP
0x50	R8
0x58	R9
0x60	R10
0x68	R11
0x70	R12
0x78	R13
0x80	R14
0x88	R15
0x90	IA32_PERF_GLOBAL_STATUS
0x98	Data linear Address
0xA0	Data Source encoding
0xA8	Latency (core cycles)

Figure 1.5: The PEBS buffer.

not defined when the processor is not in IA-32e mode are written to zero. Bytes 0xAF:0x90 are an enhancement to the PEBS record format. The value written to bytes 0x97:0x90 is the state of the IA32_PERF_GLOBAL_STATUS register before the PEBS assist occurred. This value is written so software can determine which counters overflowed when this PEBS record was written. Note that this field indicates the overflow status for all counters, regardless of whether they were programmed for PEBS or not.

The load latency facility provides a means to characterise the average load latency to different levels of the memory hierarchy. This facility requires processor supporting enhanced PEBS record format in the PEBS buffer, see Table 1.2, where the last three fields are added for load latency. The facility measures latency from micro-operation (uop) dispatch to when data is globally observable (GO). To use this feature software must assure three conditions:

– One of the IA32_PERFEVTSELx MSR is programmed to specify the event unit MEM _INST_ RETIRED, and the LATENCY_ABOVE_THRESHOLD event mask must be specified (IA32_ PerfEvtSelX[15:0] = 0x100). The corresponding counter IA32_PMCx will accumulate event counts for architecturally visible loads which exceed the programmed latency threshold specified separately in a MSR. Stores are ignored when this event is programmed.

- The MSR_PEBS_LD_LAT_THRESHOLD MSR is programmed with the desired latency threshold in core clock cycles. Loads with latencies greater than this value are eligible for counting and latency data reporting. The minimum value that may be programmed in this register is 3 (because the minimum detectable load latency is 4 core clock cycles).

- The PEBS enable bit in the IA32_PEBS_ENABLE register is set for the corresponding IA32_PMCx counter register. This means that both the PEBS_EN_CTRX and LL_EN_CTRX bits must be set for the counter(s) of interest. For example, to enable load latency on counter IA32_PMC0, the IA32_PEBS_ENABLE register must be programmed with the 64-bit value 0x00000001.00000001.

When the load-latency facility is enabled, load operations are randomly selected by hardware and tagged to carry information related to data source and latency. Latency and data source information of tagged loads are updated internally. When a PEBS assist occurs, the last update of latency and data source information are captured by the assist and written as part of the PEBS record. Loads are randomly tagged to collect latency data. The number of tagged loads with latency information that will be written into the PEBS record field by the PEBS assists can be controlled. The load latency data written to the PEBS record will be for the last tagged load operation which was retired just before the PEBS assist was invoked. The load-latency information written into a PEBS record (see Table 1.2, bytes 0xAF:0x98) consists of:

- Data Linear Address: This is the linear address of the target of the load operation.

- Latency Value: This is the elapsed cycles of the tagged load operation between dispatch to GO, measured in processor core clock domain.

- Data Source: The encoded value indicates the origin of the data obtained by the load instruction. The encoding is shown in Table 1.3. In the descriptions, local memory refers to system memory physically attached to a processor package, and remote memory referrals to system memory physically attached to another processor package.

Table 1.3: Data Source Encoding for Load Latency Record.

Encoding	Description
0x0	Unknown L3 cache miss
0x1	Minimal latency cache hit. This request was satisfied by the L1 data cache.
0x2	Pending core cache HIT. Outstanding core cache miss to same cache-line address was already underway.
0x3	This data request was satisfied by the L2.
0x4	L3 HIT. Local or Remote home requests that hit L3 cache in the uncore with no coherency actions required (snooping).
0x5	L3 HIT. Local or Remote home requests that hit the L3 cache and was serviced by another processor core with a cross core snoop where no modified copies were found. (clean).
0x6	L3 HIT. Local or Remote home requests that hit the L3 cache and was serviced by another processor core with a cross core snoop where modified copies were found. (HITM).
0x7	Reserved
0x8	L3 MISS. Local homed requests that missed the L3 cache and was serviced by forwarded data following a cross package snoop where no modified copies found. (Remote home requests are not counted).
0x9	Reserved
0xA	L3 MISS. Local home requests that missed the L3 cache and was serviced by local DRAM (go to shared state).
0xB	L3 MISS. Remote home requests that missed the L3 cache and was serviced by remote DRAM (go to shared state).
0xC	L3 MISS. Local home requests that missed the L3 cache and was serviced by local DRAM (go to exclusive state).
0xD	L3 MISS. Remote home requests that missed the L3 cache and was serviced by remote DRAM (go to exclusive state).
0xE	I/O, Request of input/output operation.
0xF	The request was to un-cacheable memory.

The layout of MSR_PEBS_LD_LAT_THRESHOLD is shown in Figure 1.6. Bits 15:0 specify the threshold load latency in core clock cycles. Performance events with latencies greater than this value are counted in IA32_PMCx and their latency is reported in the PEBS record. Otherwise, they are ignored.

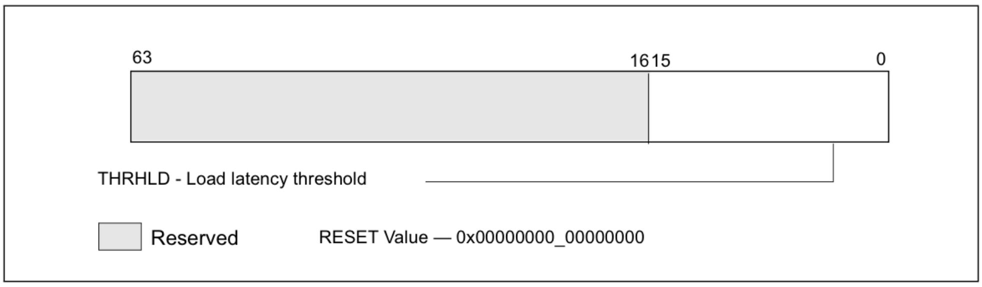

63
1615
0
THRHLD - Load latency threshold
Reserved
RESET Value — 0x00000000_00000000

Figure 1.6: Layout of MSR_PEBS_LD_LAT MSR.

The overhead from using PEBS comes from having to record in a buffer the state of the core each time it is sampled, with an extra cost for memory operations, due to latency and data source recording. As such, the overhead is mainly determined by the sampling rates: the higher the desired resolution, the larger the overhead. For most cases it can be kept below 1% while still providing enough information, specially when sampling cache misses [5, 82]. In Section 3.4.1 a study of the overhead incurred in our use of PEBS is presented.

1.3.3 Floating Point overcounting

In the Intel Sandy Bridge and following Ivy Bridge architectures, floating point operations counters count executed operations, not retired operations [86]. As a consequence, if a FP_OP is issued, but its operands are not in the cache or registers, it is counted as if it was executed, and will be reissued until its operands are stored in the cache. This means that, in cases when main memory is accessed very aggressively, floating point operations can be counted in excess, and, as a consequence, hardware counters may not be accurate. In Section 3.4.2, this issue is explained in more detail and an example is provided.

1.4 Recap

As described in Section 1.1, this work focused on computers consisting of tightly coupled processors that share memory through a shared address space. These systems fall broadly into two categories: SMP and NUMA systems The SMPs share a single centralised memory that all processors have equal access to, but in NUMA systems processors have physically distributed memory. In NUMA systems, the access time depends on the location of a data word in memory, so a processor can access its own local memory faster that a non-local one. In this way communicating data among processors becomes somewhat more complex. Modern OS allow to define an affinity among cores and memories to improve data locality and memory latency. Moving threads close to where their data reside can increase performance.

In Section 1.2, IA-64 and Intel 64 architectures were presented, and the specific computer systems used throughout this work were described.

In Section 1.3, Hardware counters were presented. Hardware counters in Intel processors allow to obtain detailed information about the memory accesses with low overhead. Sampling memory operations using EARs in Itanium 2 and PEBS in Sandy Bridge, the data memory address and the memory latency of the operations, among other information, can be recorded.

Chapter 2

Analysis of memory accesses in SMPs

The behaviour of memory accesses is one of the most significant aspects influencing the performance of any code. This fact is more and more relevant as the memory wall increases [22]. One area where the memory management and utilisation is specially important is that of parallel and distributed systems and, in particular, in current multiprocessor/multicore architectures.

For a parallel code to be correctly and efficiently executed, its programming must be careful. Taking into account architectural features, particularly the behaviour of memory accesses, is critical to improve locality among accesses and affinity of threads to cores to improve data accesses. Understanding the performance of a program requires considering several factors, such as the underlying system or the type of workload, which can lead to bottlenecks, or parts of the code where most of the time is spent. These parts can be identified by collecting information related to how the program or the system performs when executing. This collection is known as *performance monitoring*. Characterising the nature and cause of the bottlenecks using this information allows us to understand why a program behaves in a particular way.

2.1 Performance monitoring

Performance monitoring is particularly important in modern multiprocessor and multicore systems. The interplay of the cache coherency and consistency, memory hierarchies, buses, and processors is fairly complicated and far from being intuitive. Sophisticated techniques must be used to characterise the behaviour of an application executed in such systems. While profiling has been typically used to collect data in order to find out what is going wrong, another not-so-widespread use consists of using that information to take effective decisions either in runtime or in a pre-execution stage. Some performance issues in which this information is important are, among others, data locality or load balancing. Characterising them may help to improve performance.

As it was introduced in Chapter 1, HC are a powerful monitoring mechanism included in the PMU [60] of most modern microprocessors. Their use is gaining popularity as an analysis and validation tool for profiling. Their effect in the monitored program is virtually imperceptible and their precision has noticeably increased thanks to the new PEBS [28, 69] features. However, although the PMU can harvest useful information, it is not always exploited to the fullest of its capabilities. Indeed, the lack of standard tools and libraries to program HC keeps them restricted to very specific issues. Therefore, many of the possibilities that they offer have not been fully exploited yet.

There are some tools available for instrumentation and performance analysis of parallel programs. The most prevalent approach taken by these tools is to collect performance data during program execution and then provide post-mortem analysis and display performance information [58]. Important examples are SvPablo [12], TAU [77], Paradyn [68], Pin [51], or HP Caliper [24], among others. SvPablo can insert instrumentation code automatically and provide performance data for numerous metrics. TAU is capable of gathering performance information through instrumentation of functions, methods, basic blocks, and statements. The Paradyn tool leverages a technique called dynamic instrumentation to obtain performance profiles of unmodified executables. Finally, Pin and HP Caliper analyse the code at the instruction level by the use of dynamic compilation to instrument executables while they are running. They insert new instructions into the code around the instructions of the target code, which is the cause for important slowdowns.

In most cases, there is no need for precise information about the events captured, so sampled information is enough. Sampling based tools such as gprof [17] and hprof [65] are implemented by halting program execution at pre-specific times. The use of these tools implies recompiling the codes and they usually present high overheads.

Almost all of the above tools generate huge amount of profile trace-based data, which is hard to manage and manipulate. Some of them add instrumentation to the source code during the build process. Source code instrumentation may affect compiler transformations, and incurs in large overheads. Usually the execution time of the instrumented code is several times higher than the non-instrumented counterpart. In addition, they usually introduce side effects, like polluting call stacks, changes in the cache behaviour, etc., into the profiled program when adding or removing instrumentation points [10]. For example, the VTune [27] call path profiler uses binary instrumentation that fails to measure functions in stripped object code and imposes enough overhead so that Intel explicitly discourages program-wide measurement [1].

These effects are minimised in the instrumentation tool introduced in this work, which implements a much simpler method without necessarily generating trace data. Thus, if the analysis of memory behaviour of OpenMP codes is the main goal, this implementation does it with minimal effort and low overhead.

In this chapter, a set of tools that use HC to monitor the memory usage of sequential or parallel C programs is presented. This set is composed of two mostly independent parts. The first one is a instrumentation and data capture tool that, on one hand, gathers memory access information of binary codes when executing and, on the other hand, allows to insert in the user code, in a simple and transparent way, the code needed to monitor and manage HC. This way, data about the memory performance of the code can be gathered. The second one, a visualisation tool, takes the information gathered by the monitored parallel code, processes it and, finally, displays this information graphically. These tools allow users to finely tune their programs according to the use they make of the memory hierarchy.

A research field where data locality is of paramount importance is the case of irregular codes. One of the most significant group of irregular codes is composed by those ones in which indirections prevent from identifying, at compile time, the set of memory accesses performed by the application. Examples of these accesses are those performed by pointers, the indirection arrays whose content is unknown in compile time, or the use of external functions that access memory and whose structure cannot be determined a priori. Irregular codes present low locality and, due to the unpredictability of their accesses, their effective reuse of the memory is scarce. Hence, the memory hierarchy is the most important bottleneck for the efficient execution of most of these codes, being one of the issues where performance can be improved [70, 72]. In this chapter, we have used the sparse matrix vector product (SpMxV) as an example of irregular code [19], using our tools to study the behaviour of its memory accesses [40, 41]. SpMxV is the core of many important scientific applications, and several widespread techniques to improve their data locality exist in the literature [71, 13, 20].

The presented tools can also be very useful for regular codes. Another case study considered in this chapter is the vector dot product (SDOT), a regular kernel present in many scientific and engineering applications.

2.2 Information capture with HC

Gathering the information about memory accesses from EAR counters or PEBS is not as straightforward task as interacting with other kinds of HC. Instead of just reading a simple value from a few counters, there is the need to process a data buffer at irregular intervals during the execution of a program. Moreover, when running in parallel several threads or processes, filtering the information to be sure that all the relevant data are gathered in each core can become a complex task, specially if threads or processes migrate during the execution. To deal with these issues, two tools to automate this task were implemented, a data capture tool and an instrumentation tool.

2.2.1 Data capture tool

This is a command line tool [43, 42] that can work alongside any compiled binary and gathers its memory access information during its execution. The data capture tool accepts three options: the event to monitor (i.e. cache misses or TLB misses, in case of EARs; load or store instructions in case of PEBS), the sampling period, and the minimum load latency from which events are captured. This tool samples from all the events in the system, regardless of the process that originated them or the core where they took place, and saves them in a file. This file can then be read by the visualisation tool, described in Section 2.3.1, which discriminates the ones that belong to the monitored program.

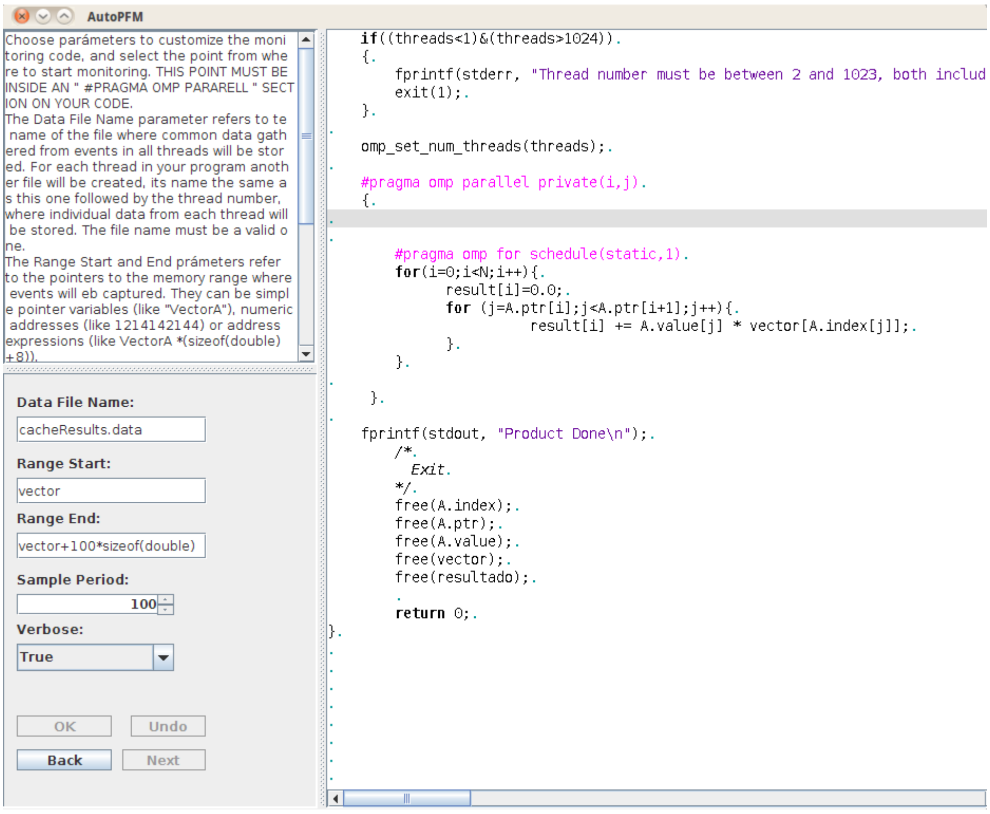
2.2.2 Instrumentation tool

If a more detailed monitoring is needed, i.e. only some parts of a code or a limited data range, instead of the entire virtual memory space, and the source code to monitor is available, the instrumentation tool [43, 42] can be used. This tool helps us to directly add the PEBS or EAR monitoring code to a target source code so that the program can monitor itself. The user only needs to indicate, through directives in the source code, where the data capture must start and end. These directives indicate the event to sample, the sampling rate, the minimum load latency, and the data range, indicated either by variable names or memory addresses (currently, this functionality is only available on Itanium). Afterwards, our instrumentation tool parses the source code and inserts the necessary monitoring code automatically.

A second version of the instrumentation tool was implemented, which includes a graphical interface designed to simplify the annotation of the code to be monitored. The reason to implement a GUI enabled version of this tool was essentially to simplify its use by non specialised users or by those not experienced with HC or even parallel programming. This version includes a text editor that allows the user to indicate directly in the source code where to insert the monitoring code. The GUI also automatically prevents the use of illegal arguments. Figure 2.1 shows the argument selection screen, where the user chooses both the initial point to start monitoring and its parameters.

The instrumentation tool uses an XML file where the monitoring code which is added to the source code under analysis, can be edited. This way, if changes are made to the machine executing the program, or new versions of the libraries are available, the monitoring code can be updated, making the tool easily portable.

The monitoring code added by the tool is placed at different points in the target code:



Choose parámeters to customize the monitoring code, and select the point from where to start monitoring. THIS POINT MUST BE INSIDE AN " #PRAGMA OMP PARARELL " SECTION ON YOUR CODE.
The Data File Name parameter refers to te name of the file where common data gathered from events in all threads will be stored. For each thread in your program another file will be created, its name the same as this one followed by the thread number, where individual data from each thread will be stored. The file name must be a valid one.
The Range Start and End prámeters refer to the pointers to the memory range where events will eb captured. They can be simple pointer variables (like "VectorA"), numeric addresses (like 1214142144) or address expressions (like VectorA *(sizeof(double)+8)).

Data File Name: cacheResults.data

Range Start: vector

Range End: vector+100*sizeof(double)

Sample Period: 100

Verbose: True

```
    if((threads<1)&(threads>1024))
    {
        fprintf(stderr, "Thread number must be between 2 and 1023, both includ
        exit(1);
    }

    omp_set_num_threads(threads);

    #pragma omp parallel private(i,j)
    {


        #pragma omp for schedule(static,1)
        for(i=0;i<N;i++){
              result[i]=0.0;
              for (j=A.ptr[i];j<A.ptr[i+1];j++){
                        result[i] += A.value[j] * vector[A.index[j]];
              }
        }

     }

    fprintf(stdout, "Product Done\n");
        /*
          Exit
        */
        free(A.index);
        free(A.ptr);
        free(A.value);
        free(vector);
        free(resultado);

        return 0;
}
```

Figure 2.1: Instrumentation tool. Parameter selection screen.

– Previous Code: This code precedes both counters programming and their reading. It includes the libraries and defines several needed constants to run the code. It also includes a number of variables and global functions (mainly for reading the EARs) to be used afterwards.

– Begin Code: This code is included in the program just before the beginning of the section to be monitored. It includes both the `libpfm2` library initialisation and the PMU registers programming, and finally the order to trigger sampling.

– End Code: This code finishes the monitoring, so it follows the section of the program being monitored. It processes the information obtained by the sampling.

Fig. 2.2 shows how the user annotations would be included in a typical situation. The tool substitutes these annotations for the appropriate monitoring code.

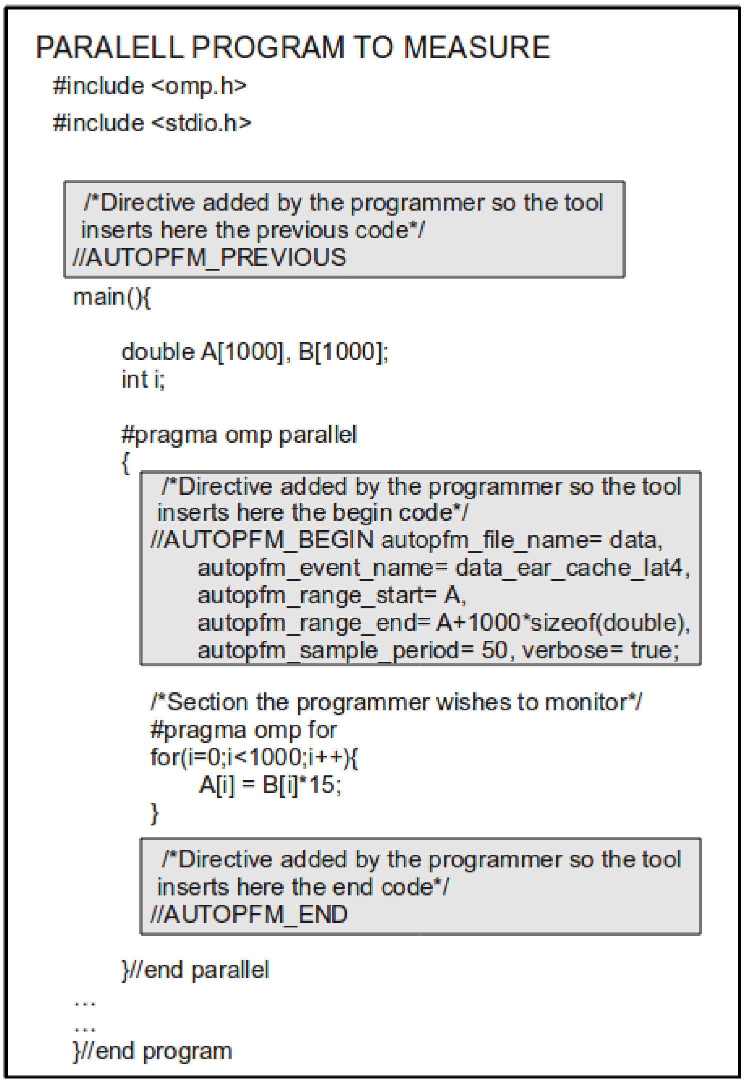
Using hardware counters in a SMP system presents several drawbacks, one of which is the need to program each of the counters present in each core. Thus performing accurate readings or measurements may be complicated and costly. Additionally, when running several threads or processes, it is important to identify on which core will they be located, or even if they will be migrated during execution.

When all processors are identical, and therefore with the same PMU, accessing the HC is simpler, as they can be accessed in the same way. Note that monitoring in one core may be performed from threads executing in different ones. Nevertheless, this complicates both programming and tracking, because it becomes harder to know exactly which code segment is being monitored. Therefore, it is simpler for each thread to monitor itself. This way, identifying which CPU and which thread executes a particular code segment becomes easier. Although less complex, this self-monitoring has several drawbacks. The most obvious one is the fact that the application performance decreases, since each thread uses some of its time for monitoring purposes. Anyway, this overhead is usually very low. In Section 3.4.1 a study of the overhead of an enhanced version of this data capture tool is performed.

The instrumentation tool presented here solves the problem of adding the necessary monitoring code, making the use of HCs (including PEBS) in programs an almost automatic task. This automation is not complete, as the user must still indicate, through simple directives, the event to capture or the starting point of monitoring in the code, but it greatly simplifies the task. For each selected event, the tool automatically adds the necessary code to access the HCs.

2.3 Data visualisation

The data gathered by the monitoring tool have to be processed and visualised to be useful. First, because of the large number of events that can be captured during the execution of a program and, second, since the interesting information is obtained by studying the data as a whole, not each event by itself. With the data obtained from the HC, the program memory



PARALELL PROGRAM TO MEASURE

```
 #include <omp.h>
 #include <stdio.h>

   /*Directive added by the programmer so the tool
   inserts here the previous code*/
  //AUTOPFM_PREVIOUS
  main(){

     double A[1000], B[1000];
     int i;

     #pragma omp parallel
     {
       /*Directive added by the programmer so the tool
       inserts here the begin code*/
      //AUTOPFM_BEGIN autopfm_file_name= data,
          autopfm_event_name= data_ear_cache_lat4,
          autopfm_range_start= A,
          autopfm_range_end= A+1000*sizeof(double),
          autopfm_sample_period= 50, verbose= true;

      /*Section the programmer wishes to monitor*/
      #pragma omp for
      for(i=0;i<1000;i++){
          A[i] = B[i]*15;
      }

       /*Directive added by the programmer so the tool
       inserts here the end code*/
      //AUTOPFM_END

     }//end parallel
  …
  …
  }//end program
```

Figure 2.2: Simple parallel vector initialisation annotated program.

hierarchy accesses can be studied in a comprehensive way. To carry out this kind of studies, different monitoring, processing and visualisation tools [77, 61, 35, 66, 27] can be used, as well as statistical or mathematical tools like Matlab, Octave, or R [53, 18, 73]. Nevertheless, it is more convenient to use a simpler tool specialised in the memory access problem, which allows a fast and general view of the results, allowing, at the same time, more detail if needed. A tool like this avoids the need to use complex and potentially difficult applications, while its specialisation makes it easier to adapt its functionality to the task at hand, simplifying its use.

2.3.1 Visualisation tool

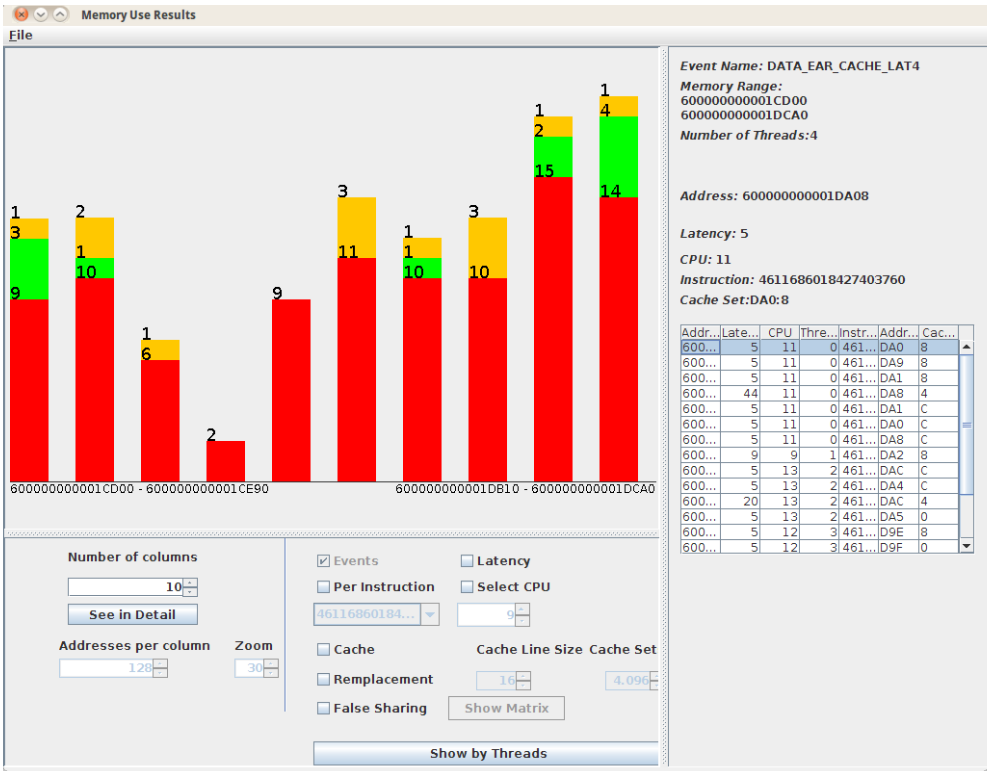
The visualisation tool [43, 42] presented here allows to classify captured events into categories according to their memory address or the thread. These events can be either cache misses or TLB misses. It shows captured events jointly in a histogram, which can be delimited by the initial and final addresses of the studied virtual memory range, or with a category for each thread.

With this tool, the data obtained by the HC can be used to analyse cache behaviour. When these data come from EARs, the tool decides whether a miss was resolved in L2 cache, L3 cache or the main memory, given the latency of the access. Note that, for PEBS the precise source of the data can be obtained directly (according to Table 1.3).

Using the captured information, several types of histograms can be displayed: Occurrence Histograms, Latency Histograms, Cache Histograms, and False Sharing or Replacement Histograms.

Occurrence Histograms show the amount of individual event instances grouped by the memory address they reference. For each histogram bar, different colors show the cache level where the miss was resolved, or, if they are TLB events, the TLB level where the address translation took place. For example, Figure 2.3 shows the number of cache misses resolved in more than 4 cycles. They are grouped by the memory address they referenced. Each bar represents a range of 400 consecutive addresses (this value can be modified by the user). Each bar in the histogram is divided in coloured sections depending on the number of misses in each level of the memory hierarchy.

Latency Histograms show the mean access latency of a group of memory addresses or


Memory Use Results
File
1
3
9
2
1
10
1
6
2
9
3
11
1
1
10
3
10
1
2
15
1
4
14
600000000001CD00 - 600000000001CE90
600000000001DB10 - 600000000001DCA0

Event Name: DATA_EAR_CACHE_LAT4

Memory Range:
600000000001CD00
600000000001DCA0

Number of Threads:4

Address: 600000000001DA08

Latency: 5

CPU: 11

Instruction: 4611686018427403760

Cache Set:DA0:8

Addr...	Late...	CPU	Thre...	Instr...	Addr...	Cac...
600...	5	11	0	461...	DA0	8
600...	5	11	0	461...	DA9	8
600...	5	11	0	461...	DA1	8
600...	44	11	0	461...	DA8	4
600...	5	11	0	461...	DA1	C
600...	5	11	0	461...	DA0	C
600...	5	11	0	461...	DA8	C
600...	9	9	1	461...	DA2	8
600...	5	13	2	461...	DAC	C
600...	5	13	2	461...	DA4	C
600...	20	13	2	461...	DAC	4
600...	5	13	2	461...	DA5	0
600...	5	12	3	461...	D9E	8
600...	5	12	3	461...	D9F	0

Number of columns: 10

See in Detail

Addresses per column: 128 Zoom: 30

☑ Events ☐ Latency

☐ Per Instruction ☐ Select CPU

46116860184... 9

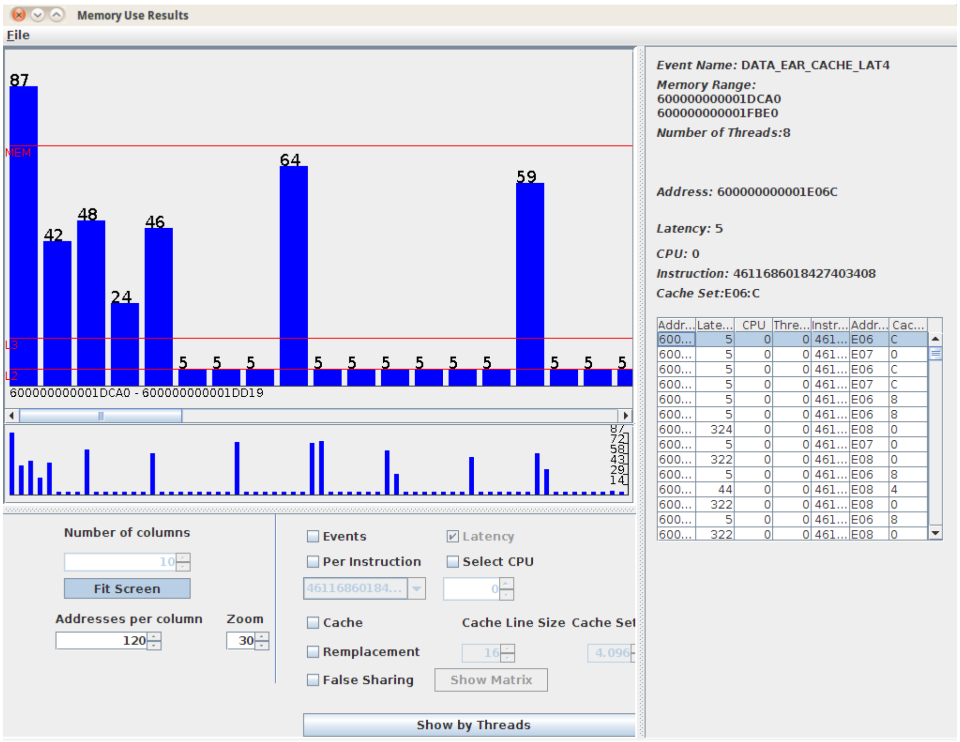
☐ Cache Cache Line Size Cache Set

☐ Remplacement 16 4.096

☐ False Sharing Show Matrix

Show by Threads

Figure 2.3: General Occurrence histogram, showing total number of cache misses detected. In the histogram L2 misses are shown in red, L3 in green, and main memory in orange.



Event Name: DATA_EAR_CACHE_LAT4
Memory Range:
600000000001DCA0
600000000001FBE0
*Number of Threads:*8

Address: 600000000001E06C

Latency: 5

CPU: 0

Instruction: 4611686018427403408

*Cache Set:*E06:C

Addr...	Late...	CPU	Thre...	Instr...	Addr...	Cac...
600...	5	0	0	461...	E06	C
600...	5	0	0	461...	E07	0
600...	5	0	0	461...	E06	C
600...	5	0	0	461...	E07	C
600...	5	0	0	461...	E06	8
600...	5	0	0	461...	E06	8
600...	324	0	0	461...	E08	0
600...	5	0	0	461...	E07	0
600...	322	0	0	461...	E08	0
600...	5	0	0	461...	E06	8
600...	44	0	0	461...	E08	4
600...	322	0	0	461...	E08	0
600...	5	0	0	461...	E06	8
600...	322	0	0	461...	E08	0

Number of columns: 10

Addresses per column: 120
Zoom: 30

□ Events ☑ Latency
□ Per Instruction □ Select CPU
46116860184... 0
□ Cache
Cache Line Size: 16
Cache Set: 4.096
□ Remplacement
□ False Sharing

Figure 2.4: Detailed Latency histogram, showing average latency of memory loads.

threads, in cycles. In this case, horizontal lines show, for each level of the memory hierarchy, the minimum latency for a read in the level. For example, in Figure 2.4, lines for the L2 cache, the L3 cache, and main memory are shown.

The tool can show the mapping of the captured events in each cache level in the so called Cache Histogram. Individual event instances are grouped according to their cache set. The cache line size and the number of cache sets can be chosen by the user in the graphical interface. When these parameters are set to the actual values of a cache level of the studied system, an accurate view of the execution is obtained. The visualisation tool can also simulate different kinds of cache memories, and it can show how the program data would be mapped in other systems. This way different cache architectures can be studied with data traced from just one execution of the code. Note that latency data depend on the executing system, so the simulated caches do not include latency histograms.

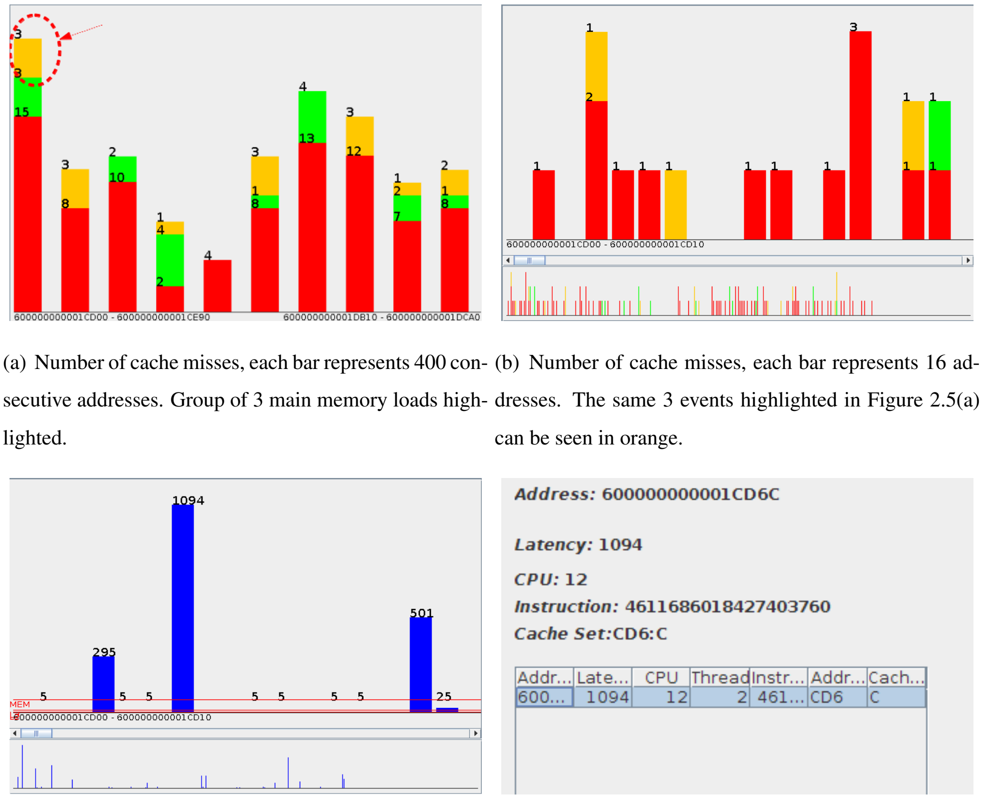
False Sharing or Replacement Histograms show conflicts among threads in a cache line. The tool can extract information about the cache replacements or possible instances of false sharing [81]. The replacement histogram may be useful in cases where some threads share the same cache level, because it shows the instances of accesses to the same cache set by different threads. So, this histogram shows how a different data partitioning among threads may affect the cache usage. False sharing can arise in systems with distributed, coherent caches: if two processors operate with different data stored in the same cache line, a write operation in one of them may force the whole cache line to be invalidated. The visualisation tool offers a histogram showing the instances where two different CPUs access different data in the same cache line.

When showing any of these histograms by address, both a General and a Detailed view are available. The General view shows the complete memory range under study. The number of categories in the histogram can be modified (by clicking in the 'Number of Columns' spinner shown in Figure 2.3). The Detailed view shows the whole studied memory range, adjusting the number of addresses making up each category, down to just one address. If the memory range does not fit inside the application window, a scroll can be used to move through the range, and a small general histogram is used to help navigation. (See Figure 2.4.)

Figure 2.5 shows and example of how the tool helps the user to detect memory addresses with large access time. In this example, a group of 3 main memory loads can be seen in Figure 2.5(a), highlighted with an arrow. A detailed view shows how these 3 events read data from 3 different virtual memory addresses (Figure 2.5(b)). In Figure 2.5(c) the mean latency for each of the categories present in Figure 2.5(b) is shown, being higher in those with main memory loads. By clicking in any histogram bar, a table with the events in that memory range is displayed, as shown in Figure 2.5(d), which allows the user to see the details of any particular event.

2.4 Case studies

In this section, the use of the tools we propose to characterise the behaviour of the memory accesses for an irregular parallel code, the sparse matrix vector product (SpMxV, $y = A \times v$),



(a) Number of cache misses, each bar represents 400 consecutive addresses. Group of 3 main memory loads highlighted.

(b) Number of cache misses, each bar represents 16 addresses. The same 3 events highlighted in Figure 2.5(a) can be seen in orange.

(c) Latencies, each bar represents 16 addresses. The same 3 events highlighted in Figure 2.5(a) can be seen increasing mean latency in their respective categories.

Address: 600000000001CD6C

Latency: 1094

CPU: 12

Instruction: 4611686018427403760

Cache Set:CD6:C

Addr...	Late...	CPU	Thread	Instr...	Addr...	Cach...
600...	1094	12	2	461...	CD6	C

(d) Event table.

Figure 2.5: Individual cache misses.

and a regular dense SDOT product ($y = x{\cdot}y$), are shown. These codes were parallelised using OpenMP [64]. The SpMxV was executed in our Itanium Server (see 1.2 for more details). Remember that in this system, cores 0 to 7 have greater affinity to one half of the main memory (local to them) and cores 8 to 15 to the other half (remote to cores 0 to 7).

The study for the SDOT was carried out in our Xeon Server X (see 1.2.2 for more details).

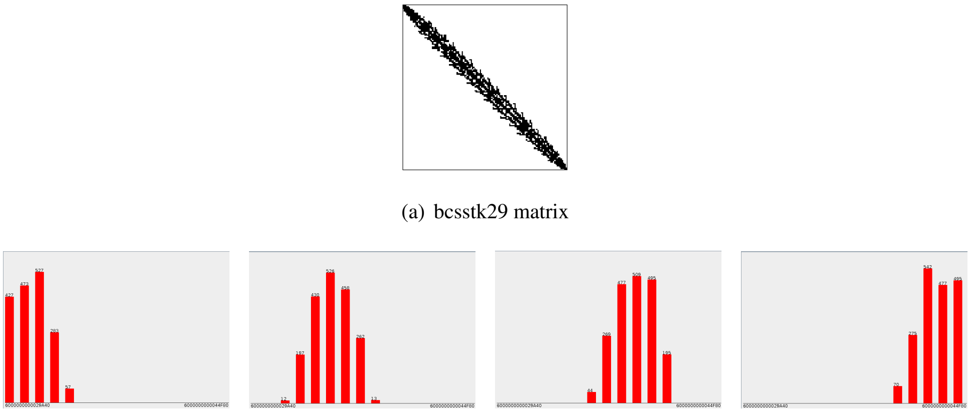
2.4.1 Sparse Matrix Vector Product

The main goal of this study was to test how, by using EAR counters and the developed tools, a useful picture of the behaviour of shared memory accesses of a parallel program can be obtained. In this study, we have executed the SpMxV with several double precision floating point sparse matrices and different numbers of threads, capturing EAR events in each execution.

The parallelisation of the SpMxV can be made by distributing either the rows or columns, or both, of the matrix among the processors. In the implementation under study, the matrix is evenly distributed by rows in a block fashion, and each thread calculates a slice of the result vector. When working with sparse matrices, the irregularities in data placement make the distribution among threads unbalanced. This way, the cores with a larger workload usually present more cache misses. Both the locality and affinity of the threads and data can be analysed with our tools.

The event capture begins just before the main matrix vector product, and ends just after its termination. Events that take place during vector initialisation or matrix reading are not captured. How OpenMP shares work among threads can be checked easily, and in particular, how the dispersion of the data affects matrix operations and how the processor PMU captures events.

In the Itanium 2 family, accesses to floating point data do not use the L1D cache, but they are read from the L2D directly. Therefore, as vector v stores floating point data, all read accesses to v will give rise to a cache miss with a latency greater than 4 cycles, and they can be captured by the EARs. Only the sampling procedure imposed by the PMU to read and write these counters limits the percentage of accesses that can be detected. Note that all accesses to v are guided by the indirection given by the sparse matrix A pattern, as the SpMxV code



(a) bcsstk29 matrix

(b) Thread 0

(c) Thread 1

(d) Thread 2

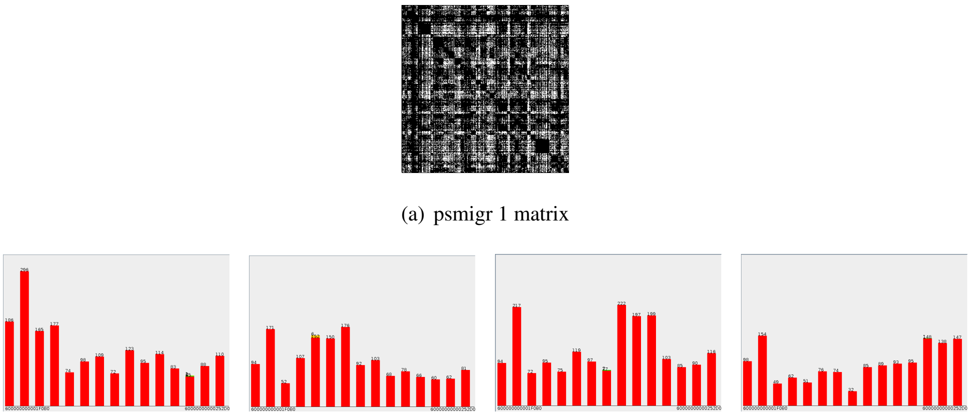
(e) Thread 3

Figure 2.6: Number of cache misses by address, the x axis represents the memory range of vector v. Distribution of the bcsstk29 matrix in 2.6(a).

accesses only the positions in v that match nonzero entries in the rows.

Figures 2.6 and 2.7 show the amount of captured read accesses to v made by each thread, for two different matrices. Each histogram shows, for a given v address range, the number of captured events. In this example, the data have been gathered using the instrumentation tool, with a 50 events sampling period. These figures show how the matrix pattern influences the dispersion of read accesses. Because the *bcsstk29* matrix (Figure 2.6(a)) has a band pattern, accesses to v from each of the 4 threads are gathered in different memory areas as it can be seen in Figures 2.6(b) to 2.6(e). On the other hand, since the *psmigr_1* matrix (Figure 2.7(a)) is more homogeneous than *bcsstk29*, memory is shared uniformly among threads (Figures 2.7(b) to 2.7(e)). Note that there is a small band component in this matrix, as a rise of accesses in that zone from each thread shows.

Figure 2.8 shows the latencies gathered from executing the SpMxV with 4 threads using the *sme3Da* matrix (Figure 2.8(a)). Due to the pattern of this particular matrix, work is distributed among threads quite evenly. In this example, the threads were forced to be in different cores of two cells (threads 0 and 1 run in cores 0 and 1, in cell 1, whereas threads 2 and 3 run in cores 8 and 9, in cell 0). Moreover, core 0 is imposed to take care of reading



(a) psmigr 1 matrix

(b) Thread 0

(c) Thread 1

(d) Thread 2

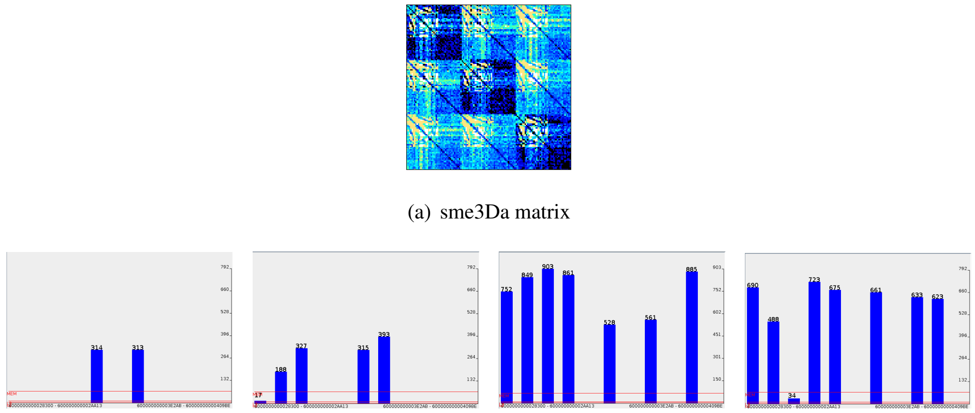
(e) Thread 3

Figure 2.7: Number of cache misses by address, the x axis represents the memory range of vector v. Distribution of the psmigr_1 matrix in 2.7(a).

both the matrix and the vector from file, so they are stored in the local memory of cores 0 to 7. This means that core 0 has its cache preloaded, so it shows fewer accesses to main memory (Figure 2.8(b)). Core 1 needs to fill its L2 cache, so it presents more accesses to main memory than core 0. Finally, cores 8 and 9 are placed in a different cell, so all their accesses to main memory take more time, as depicted in Figures 2.8(d) and 2.8(e). This is a graphical representation of the effects of the affinity between cores and memory accesses. Latencies of more than 16 cycles are those usually served by the main memory, or, in some cases, by the L3 cache.

2.4.2 Vector-vector dot product, SDOT

To illustrate capabilities of the tools to study the low levels of the cache, the SDOT operation, $y = x \cdot y$, was considered, where both x and y are single precision floating point vectors of length n. The SDOT was parallelised with a block cyclic data distribution with a block size b. Since the result is stored in vector y, there might be false sharing among cores. We have used our tools to show how the block size influences the use of cache memories and their impact on the performance. The accesses to vector y, where data is being both loaded and stored, were studied. Depending on the value of b, the use made of the lower levels of the cache is



(a) sme3Da matrix

(b) Core 0

(c) Core 1

(d) Core 8

(e) Core 9

Figure 2.8: Latencies to access main memory per core (in cycles). The x axis represents the memory range of vector v.

different. In this example, all accesses with a latency higher than 10 cycles were sampled, this means that low latency L1 accesses are ignored.

Note that the cache line size in Xeon Server X is 64 bytes long, storing 16 floating points per line. This way, if this code is executed with 8 cores and $b = 2$, there will be false sharing in vector y among the 8 cores, since they all access the same line, but different data. In Figure 2.9, L1 hits are shown in blue, L1 Line Buffer hits in red, L2 hits in green, and L3 hits in orange. This figure shows how, as the block size increases, the number of L1 cache accesses increase, since fewer cache lines are invalidated. Also, the mean access latency in the program decreases. With $b = 2$, false sharing greatly influences performance. In this case, many accesses to the L1 Cache Line Buffer are captured, as well as to the L2. This is because the cores are invalidating lines to each other, causing misses to the L1. Data can be served from higher cache levels or directly by the L1 line buffer if there are outstanding requests from other cores. With $b = 16$, false sharing is completely eliminated, so the L1 is used more efficiently, since there are fewer invalidations, but there is no use of preloaded lines. Due to a sampling artifact, more accesses to L3 are detected. This is because, as low latency L1 accesses are ignored, and with $b = 16$ their frequency increases compared to $b = 2$, the L3 accesses have a greater chance of being captured, thus higher latencies are obtained.

With $b = 32$ there is no false sharing and preloaded lines (to the L2 level) are used, so fewer accesses to L3 are needed and latency greatly decreases.

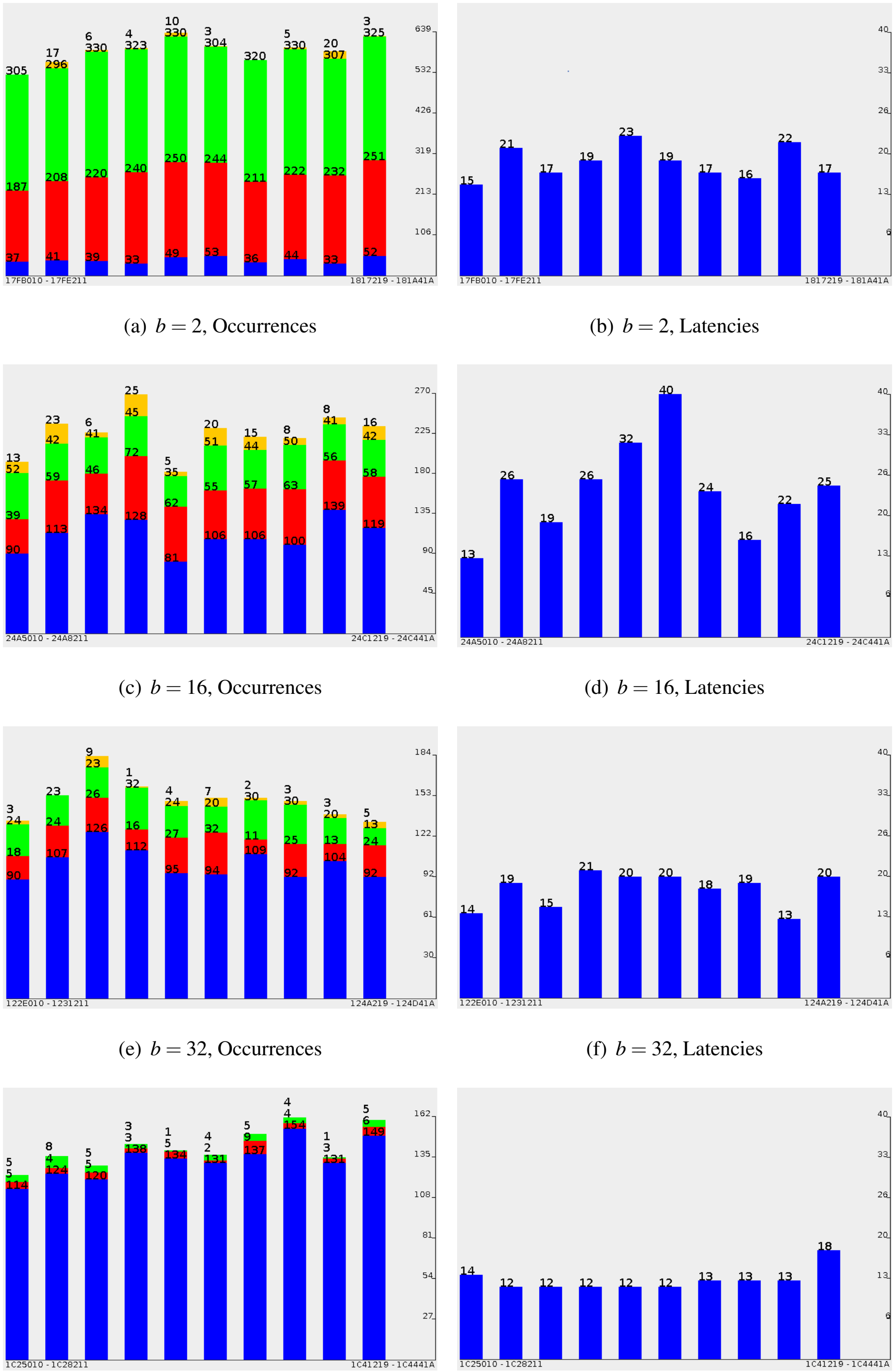
A pure block distribution shows the best results for this problem, since memory is distributed among threads in contiguous slices and thus there is no false sharing, memory can be preloaded, and the cache is used evenly.

2.5 Recap

Obtaining performance data is not a straightforward matter, as explained in Section 2.1. Although several tools exist, they present some drawbacks . Tools to simplify the process of obtaining and studying data provided by complex hardware counters have been presented. Tools presented in Section 2.2 aim to gather information about the memory addresses accessed by the threads and their latencies from these counters. The specific focus on memory accesses of our tools makes them more suitable for analysing the memory behaviour than other general purpose ones. In addition, the use of HC allows these monitoring tools to present reduced overheads. Since the added monitoring code is user editable, the tools can be adapted for their use in any number of environments, architectures or languages. The data visualisation tool presented in Section 2.3 allows the user to carry out statistical studies of the captured events, to understand the behaviour if their codes. It does so by offering the most important functionalities of hardware counters. This tool shows, in a friendly way, valuable information about locality, memory access patterns and affinity among data and cores. In particular, the information provided by the tool can be useful to analyse the influence of false sharing, load balance, coherence implications and other memory related issues.

In Section 2.4, parallel OpenMP shared memory programs were considered in to show the functionality of the tools. The parallel SpMxV problem was used as a case study in a Itanium 2 based system, showing how the proposed tools can be used for analysing the behaviour of the memory hierarchy. A regular parallel code, SDOT, was used as a different case study, to illustrate the detection of false sharing on a Xeon based system. Both examples show how the proposed tools can be useful for analysing the effects of the cache coherence mechanisms on data partitioning in parallel systems.

Manuals for the applications presented in this chapter, as well as the source code of the tools, can be found at the CiTIUS git repository [62].



(a) $b = 2$, Occurrences

(b) $b = 2$, Latencies

(c) $b = 16$, Occurrences

(d) $b = 16$, Latencies

(e) $b = 32$, Occurrences

(f) $b = 32$, Latencies

(g) Block distribution, $b = 4000$, Occurrences

(h) Block distribution, $b = 4000$, Latencies

Figure 2.9: Occurrences and latencies of cache accesses with 4 threads, $n = 32000$, $r = 100000$, and different block sizes, including a block distribution.

Chapter 3

Performance models based on runtime information

While memory accesses are an important issue influencing performance, it is not the only one. Issues dealing with the code execution, and, in parallel codes, issues of thread synchronisation like the operations issue order or the use of vector data, are also of the utmost importance. This makes measuring and modelling performance in an easy way an invaluable tool for programmers to improve their codes.

To understand the performance of a code running on a particular system, various performance models and tools have been proposed [58, 1, 59, 16, 10, 57, 76]. In [21] a mean value analysis of a multithreaded multicore processor is performed. Their results show that there is a performance valley to be avoided as the number of threads increases. Markovian models are used in [8] to model a cache memory subsystem with multithreading. Other works [4, 31] propose to model multithreaded multicore using queuing theory.

One of the most succesful models for multicore architectures is the Roofline Model (RM) [85], which offers a nice balance between simplicity and descriptiveness based on two important concepts: the operational intensity (OI) and the number of FLOPS (floating point operations per second). Nevertheless, its own simplicity might hide some performance bottlenecks present in modern architectures. Two extensions to the RM are proposed in this chapter. In the first one, the RM is extended taking different measurements during the life of an application, in order to show the evolution of different phases of the execution. This extension shows the evolution in time of a kernel, in a per thread basis. This has special importance in multicore and heterogeneous systems, since it shows clearly the differences in the execution on each core. We call this model the Dynamic Roofline Model (DyRM). In the second one, a third dimension is added to the model, showing the average latency of memory accesses for each measurement, in order to show imbalances in memory access among threads or cores. The representation of this new model is 3D, in contrast with the 2D nature of the former RM and DyRM, hence we called it 3DyRM.

Both models make use of hardware counters to collect data which is analised and visualised using a performance visualisation too that we have also developed.

3.1 Berkeley Roofline Model

The Berkeley Roofline Model [85] (RM) is an easy-to-understand model, offering performance guidelines and information about the behaviour of a program, information that can help programmers to understand the performance of their codes.

It is likely that for the recent past and foreseeable future, off-chip memory bandwidth will often be the constraining resource. Hence, a model that relates processor performance to off-chip memory traffic is needed. Towards that goal, the term operational intensity (OI) is used to mean number of floating point operations (Flops) per byte of DRAM traffic (in Flops/Byte or FlopsB). Total bytes accessed are defined as those that go to the main memory after they have been filtered by the cache hierarchy. That is, traffic is measured between the caches and memory rather than between the processor and the caches. Thus, operational intensity quantifies the DRAM bandwidth needed by a kernel on a particular computer.

In the RM, operational intensity is used instead of the terms arithmetic intensity or machine balance for two reasons. First, arithmetic intensity and machine balance measure traffic between the processor and cache, whereas the goal is to measure traffic between the caches and DRAM. This subtle change allows to include memory optimisations of a computer into this bound and bottleneck model. Second, the model will work with kernels where the operations are not arithmetic, so a more general term than arithmetic is needed.

The RM ties together floating point performance, operational intensity, and memory performance in a two-dimensional graph. Peak floating point performance can be found using the hardware specifications or microbenchmarks. If the working sets of the kernels considered do not fit fully in on-chip caches, peak memory performance is defined by the memory system behind the caches. Memory performance can be found, for example, with the STREAM benchmark [56], or other optimised microbenchmarks designed to determine sustainable DRAM bandwidth.

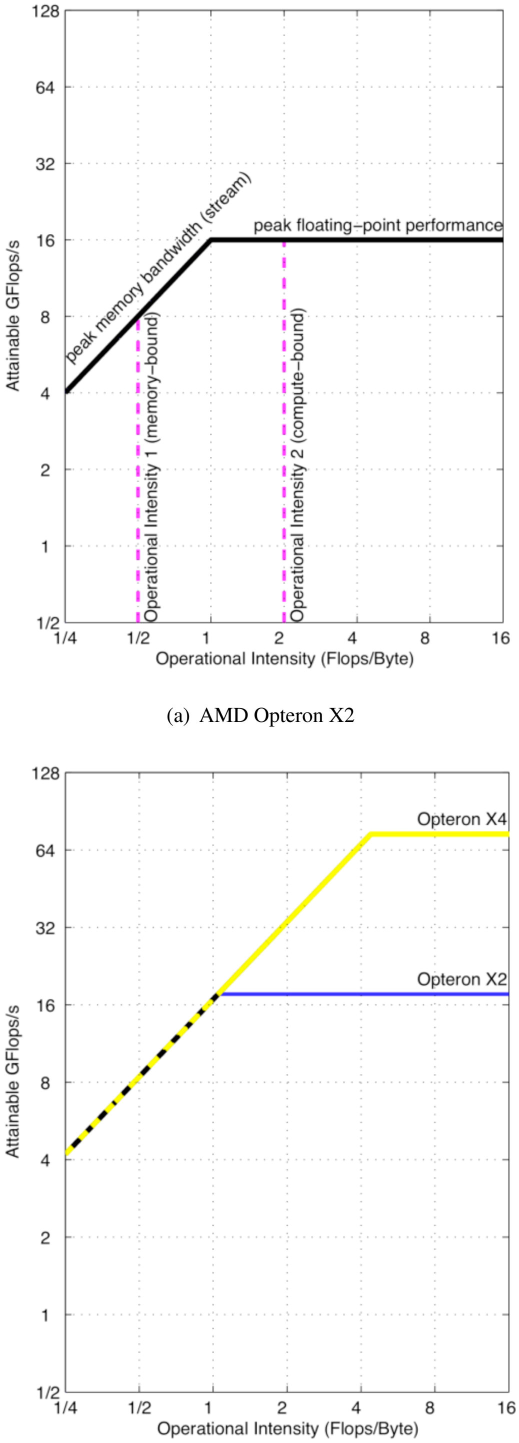
Figure 3.1(a) shows the RM for a 2.2 GHz AMD Opteron X2 model 2214 in a dual socket system. The graph is on a log-log scale. The Y-axis is attainable floating point performance. The X- axis is operational intensity, varying from 1/4 Flops/DRAM byte accessed to 16 Flops/DRAM byte accessed. The system being modelled has a peak double precision floating point performance of 17.6 GFlops and a peak memory bandwidth of 15 GBytes/sec from a memory bandwidth benchmark. Note that this latter measure is the steady state bandwidth potential of the memory in a computer, not the pin bandwidth of the DRAM chips.

A horizontal line showing peak floating point performance of the computer can be plotted. Obviously, the actual floating point performance of a floating point kernel can be no higher than the horizontal line, since that is a hardware limit. Besides, since X-axis is Flops per byte and the Y-axis is GFlops per second, bytes per second—which equals (GFlops/second)/(Flops/byte)— the peak memory performance is just a line at a 45-degree angle in this figure. Hence, a second line can be plotted that gives the maximum floating point performance that the memory system of that computer can support for a given operational intensity. Next formula drives the two performance limits in the graph in Figure 3.1(a):

Attainable GFlops = Min(Peak Floating Point Performance, Peak Memory Bandwidth × Operational Intensity)

These two lines intersect at the point of peak computational performance and peak memory bandwidth. The horizontal and diagonal lines give this bound model its name.

Note that these limits are representative of a multicore system and they are created once per multicore system, not once per kernel. For a given kernel, a point on the X-axis can be found based on its particular operational intensity. If a (pink dashed) vertical segment from



(a) AMD Opteron X2

(b) Opteron X2 vs. Opteron X4

Figure 3.1: Roofline Model for (a) AMD Opteron X2 on left and (b) Opteron X2 vs. Opteron X4 on right.

that point to the model limit is drawn, the performance of the kernel on that computer must lie somewhere along that segment.

The Roofline sets an upper bound on performance of a kernel depending on its operational intensity. If we view the operational intensity as a column that hits the roof, either it hits the flat part of the roof, which means performance is compute bound, or it hits the slanted part of the roof, which means performance is ultimately memory bound. In this example, in Figure 3.1(a), a kernel with operational intensity 2 is compute bound and a kernel with operational intensity 1 is memory bound.

Note that the ridge point, where the diagonal and horizontal roofs meet, offers an insight into the overall performance of the computer. The x-coordinate of the ridge point is the minimum operational intensity required to achieve maximum performance. If the ridge point is far to the right, then only kernels with very high operational intensity can achieve the maximum performance of that computer. If it is far to the left, then almost any kernel can potentially hit the maximum performance. The ridge point suggests the level of difficulty for programmers and compiler writers to achieve peak performance.

To illustrate these concepts, we can compare the Opteron X2 with two cores in Figure 3.1(a) to its successor, the Opteron X4 with four cores (Figure 3.1(b). To simplify board design, they share the same socket. Hence, they have the same DRAM channels and can thus have the same peak memory bandwidth, although the prefetching is better in the X4. In addition to doubling the number of cores, the X4 also has twice the peak floating point performance per core: X4 cores can issue two floating point SSE2 instructions per clock cycle while X2 cores can issue two every other clock cycle. As the clock rate is slightly faster—2.2 GHz for X2 versus 2.3 GHz for X4—the X4 has slightly more than four times the peak floating point performance of the X2 with the same memory bandwidth. Figure 3.1(b) compares the Roofline models for both systems. As expected, the ridge point shifts right from 1.0 in the Opteron X2 to 4.4 in the Opteron X4. Hence, to see a performance gain in the X4, kernels need an operational intensity higher than 1.

3.1.1 Adding ceilings

According to [36], one advantage of bound and bottleneck analysis is that:

> a number of alternatives can be treated together, with a single bounding analysis providing useful information about them all.

The Roofline model provides this bound and bottleneck analysis for performance. Suppose a code is performing far below its Roofline, we could ask what optimisations should be performed, and in what order.

This insight can be leveraged to add multiple ceilings to the Roofline model to guide which optimisations to perform, which are similar to the guidelines that loop balance gives the compiler. Each of these optimisations can be thought of as a "performance ceiling" below the appropriate Roofline, meaning that you cannot break through a ceiling without performing the associated optimisation.

For example, to reduce computational bottlenecks on the Opteron X2, two optimisations can help almost any kernel:

1. Improving instruction level parallelism (ILP) and applying SIMD. For superscalar architectures, the highest performance comes when fetching, executing, and committing the maximum number of instructions per clock cycle. One way to increase ILP is by unrolling loops. For the x86-based architectures, another way is using floating point SIMD instructions whenever possible, since an SIMD instruction operates on two or four pairs of adjacent operands.

2. Balancing floating point operation mix. The best performance requires that a significant fraction of the instruction mix be floating point operations. Peak floating point performance typically also requires an equal number of simultaneous floating point additions and multiplications, since many computers have multiply-add instructions or because they have an equal number of adders and multipliers.

To reduce memory bottlenecks, three optimisations can help:

4. Restructuring loops for unit stride accesses. Optimising for unit stride memory accesses engages hardware prefetching, which significantly increases memory bandwidth.

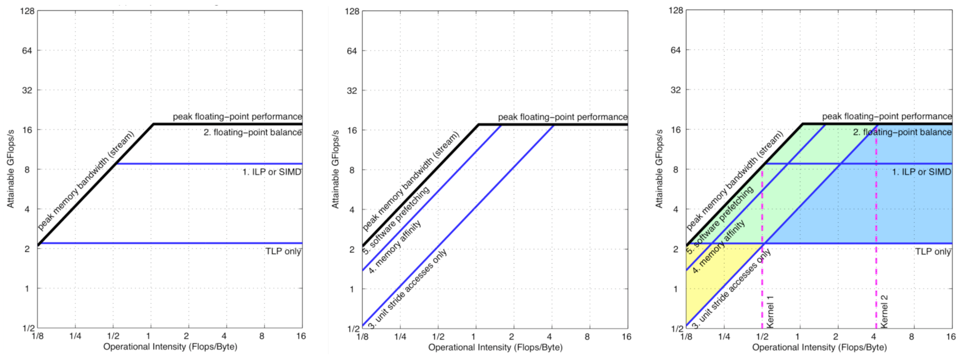
5. Ensuring memory affinity. Most microprocessors today include a memory controller on the same microprocessor. Data and the threads tasked to that data should be allocated to the same memory-processor pair, so that the processors rarely have to access the memory attached to other chips.

6. Using software prefetching. Usually the highest performance requires keeping many memory operations in flight, which is easier to do via prefetching rather than waiting until the data is actually requested by the program. On some computers, software prefetching delivers more bandwidth than hardware prefetching alone.

The computational ceilings can come from a manual optimisation, although it is easy to collect the necessary parameters from simple microbenchmarks. The memory ceilings require running experiments on each computer to determine the gap between them. The good news is that ceilings only need be measured once per multicore computer.

Figure 3.2 adds ceilings to the Roofline model in Figure 3.1. In particular, Figure 3.2(a) shows the computational ceilings and Figure 3.2(b) the memory bandwidth ceilings. Although the higher ceilings are not labelled with lower optimisations, they are implied: to break through a ceiling, it is necessary to have already broken through all the ones below it. Figure 3.2(a) shows the computational “ceilings” of 8.8 GFlops if the floating point operation mix is imbalanced and 2.2 GFlops if the optimisations to increase ILP or SIMD are also missing.

Figure 3.2(b) shows the memory bandwidth ceilings of 11 GBytes/sec without software prefetching, 4.8 GBytes/sec without memory affinity optimisations as well, and 2.7 GBytes/sec with only unit stride optimisations (all measured with the STREAM benchmark). Figure 3.2(c) combines the other two figures into a single graph. The operational intensity of a kernel determines the optimisation region, and thus which optimisations to try. The middle of Figure 3.2(c) shows that the computational optimisations and the memory bandwidth optimisations overlap. Colours were picked to highlight that overlap. For example, kernel 2 falls in the blue trapezium on the right, which suggests working only on the computational optimisations. If a kernel fell in the yellow triangle on the lower left, the model would suggest trying just memory optimisations. Kernel 1 falls into the green (= yellow + blue) parallelo-



(a) Computational Ceilings.

(b) Bandwidth Ceilings

(c) Optimisation Ceilings

Figure 3.2: Roofline Model with Ceilings for Opteron X2.

gram in the middle, which suggests trying both types of optimisations. Note that the vertical lines of kernel 1 fall below the floating point imbalance optimisation, so optimisation 2 may be skipped.

Therefore, the ceilings of the Roofline model suggest which optimisations to perform. The height of the gap between a ceiling and the next higher one is the potential reward for trying that optimisation. Thus, Figure 3.2 suggests that optimisation 1, which improves ILP/SIMD, has a large potential benefit for improving computation on that computer, and optimisation 4, which improves memory affinity, has a large potential benefit for improving memory bandwidth on that computer.

The order of the ceilings suggest the optimisation order, so the ceilings should be ranked from bottom to top: those most likely to be realised by a compiler or with little effort by a programmer are at the bottom and those that are difficult to be implemented by a programmer or inherently lacking in a kernel are at the top. The one quirk is floating point balance, since the actual mix is dependent on the kernel. For most kernels, achieving parity between multiplies and additions is very difficult, but for a few, parity is straightforward. One example is the sparse matrix-vector multiplication. For that domain, we would place floating point mix as the lowest ceiling, since it is inherent.

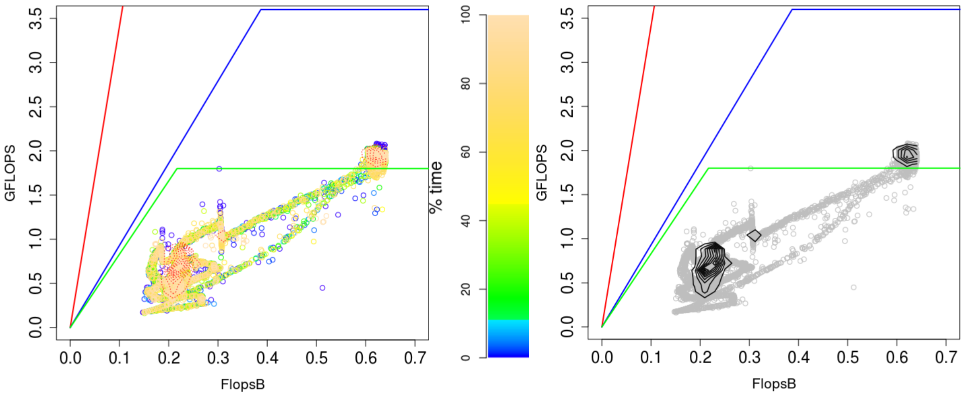
3.2 Roofline Model extensions

The RM gives a simple representation of a program performance on a particular system. Nonetheless, in some cases it may be misleading. For example, consider a program which goes through two phases of execution. One of them might be close to the maximum GFLOPS and OI of the machine, while the other is performing much poorly. The RM would place the program performance at a single point of the figure, perhaps between the performance of both phases, which would not be representative of the real program behaviour. In another example, consider a heterogeneous system. While the RM would give a performance point for the entire system, thus hiding the heterogeneity, differences inside the system would suggest that threads should have to be studied separately. Situations like these justify our proposal which provides information at regular intervals of an execution, on a thread by thread basis. This extension can be viewed as a Dynamic Roofline Model (DyRM) [44, 45].

3.2.1 Dynamic Roofline Model

Our proposal, the DyRM, is essentially the equivalent of dividing in time slices the execution of a code and getting one RM for each one, then combining them in just one graph. This way, a more detailed view of the performance during the entire life of the code, showing its evolution and behaviour, is obtained. In DyRM, linear axes are used instead of the logarithmic axes of the original RM to show more minute differences in the behaviour. As an example, Figure 3.3(a) shows the DyRM of a NAS [30] application running on an Intel Xeon E5-2603 (Xeon Server A, introduced in Section 1.2.2). A colour gradient is used to show the program evolution in time. In this way, each point in the model is coloured according to the time elapsed since the start of the program.

With this approach different execution phases or behaviours in the code can be easily detected, as can be seen in Figure 3.3. In addition, to better show the phases, a two dimensional density estimation of the points in the extended model can be performed (Figure 3.3(b) and red dotted lines in Figure 3.3(a)). Such an estimation allows us to readily find clusters, i.e., zones where the code spends more time, which are needed to identify performance bottlenecks. The resulting groups can be highlighted and, by changing the colour of the points in



(a) DyRM

(b) Density colouring

Figure 3.3: Examples of Dynamic Roofline Models for the NAS Parallel benchmark SP.B.

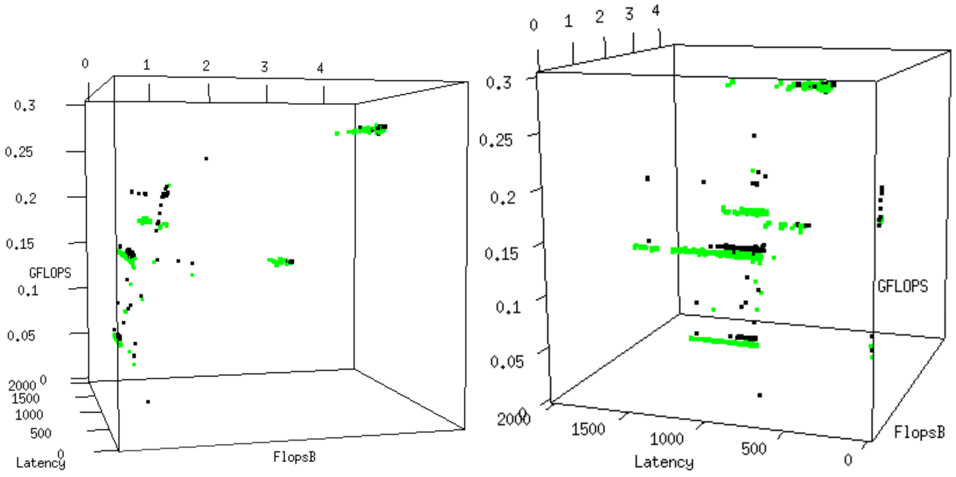
the DyRM, a better view of the clusters can be obtained. By using both graphs, we combine the simplicity of the RM with a detailed view of a program execution.

In the example of Figure 3.3, three roofs were drawn, representing the expected maximum performance of a computer core in certain situations. Note that the slope of the slanted part of the roof depends on the memory bandwidth used. The topmost roof represents the peak GFLOPS using SIMD instructions and its theoretical maximum memory bandwidth [26]. This roof reaches 14.4 GFLOPS for the Xeon E5-2603 and it is cut from the figure, because of the use of a lineal scale, as said before. The middle roof represents the maximum GFLOPS without SIMD instructions, considering one multiply and one add operations per cycle and the maximum memory bandwidth given by the STREAM benchmark [56, 55]. The lowest roof represents the GFLOPS considering only one floating operation per cycle and the worst memory bandwidth given by the STREAM benchmark. In Figure 3.3(a), it is shown that the example application remains mostly under the lower roof during its execution. Figure 3.3(b) shows two clusters where the application spends more of its execution time. One of these clusters exceeds the lower roof, meaning it makes more than one floating point operation per cycle by combining add and multiply operations. The other cluster is below the slanted side of the roof, so it is ultimately memory bound.

3.2.2 Latency Extended Dynamic Roofline Model

RM models the memory performance of a system-program using the OI. OI takes into account the cache hierarchy (since a better use of cache memories would mean less use of main memory) and the memory bandwidth and speed (since its performance would affect GFLOPS). Yet, to characterise the performance, it may be insufficient, specially on NUMA systems. The RM sets system upper limits to performance, but on a NUMA system, distance and connection to memory cells from different cores may imply variations in the memory latency. This information is valuable in many cases. Variations in access time cause different values in the GFLOPS for each core, even if each core performs the same number of operations. This way the same code may perform differently depending on where it was scheduled. In these cases, OI may remain the same, hiding the fact that poor performance is due to the memory subsystem. A programmer trying to increase the application performance would not know whether the differences in GFLOPS are due to memory access or a different reason, like power scaling or the execution of other processes in some processors. Extending the DyRM with a third dimension showing the mean latency of memory accesses for each point in the graph would clarify the source of the performance issue. We call this new model Latency Extended Dynamic Roofline Model, 3DyRM [46, 45]. As stated in Chapter 1, modern Intel processors can measure the latency of memory accesses using PEBS [28], so this model can be easily obtained for Intel systems.

Figure 3.4 shows an example of the 3DyRM for the execution of an example code, using 8 threads on a dual processor system with 8 cores and NUMA memory system. In this model, a third dimension showing the mean memory latency in number of cycles is added. Figure 3.4(a) shows the GFLOPS and OI axes, in terms of the Roofline Model with the dynamic information introduced in the previous section. Points from threads in cores of processor 0 are shown in black, while points from threads in cores of processor 1 are shown in green. Note that the example code presents different phases, and some of them result in lower GFLOPS for processor 0. In Figure 3.4(b) the same 3DyRM is shown focusing on the GFLOPS and Latency axes. Note that for processor 1, memory accesses may take longer than for processor 0. This is because, in this example, all data are stored in a memory cell closer to processor 0, so memory accesses take longer for processor 1. This way, in phases with many



(a) GFLOPS/FlopB

(b) GFLOPS/Latency (cycles)

Figure 3.4: 3DyRM. Two views for the same code, GFLOPS/FlopsB/Latency (cycles). Data from processor 0 are shown in black, those from processor 1 are in green.

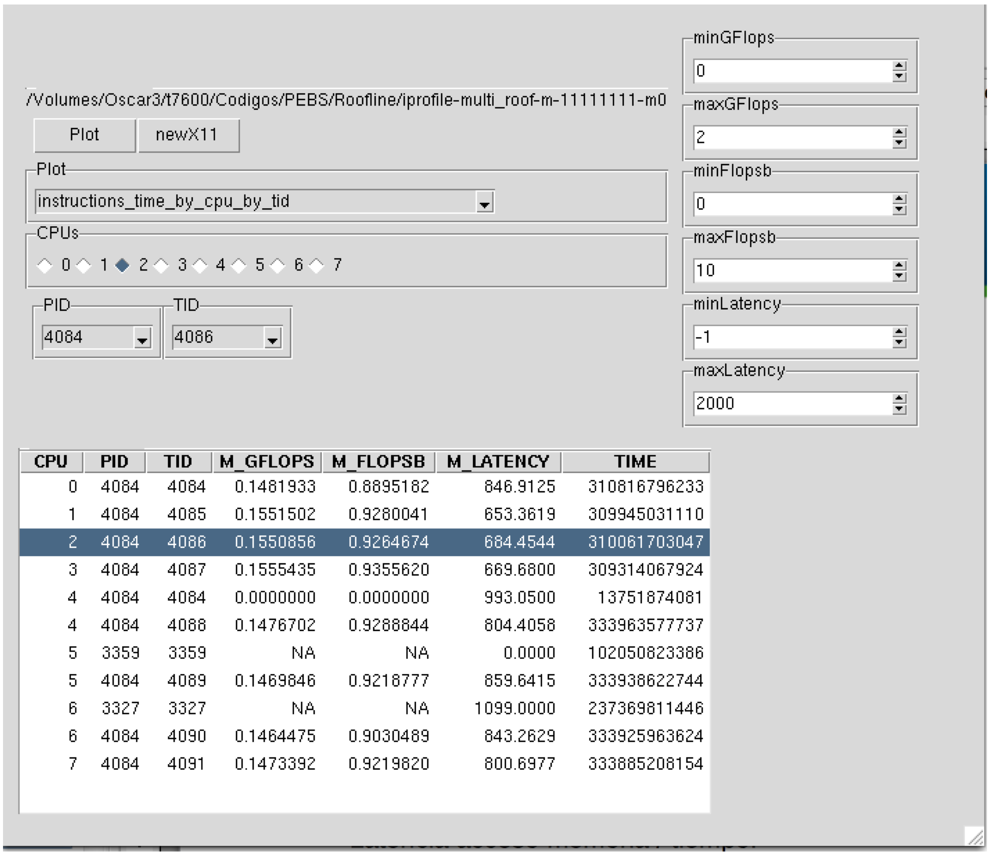
main memory accesses, the same code reaches better performance in cores of processor 0. This information, which was not clear in Figure 3.4(a), may become increasingly important for larger shared memory machines, with more cores and memory cells, and with larger differences in access time.

3.3 Performance analysis tool

In order to obtain the 3DyRM, a set of tools that collects and processes performance information of a complete shared memory system has been implemented. The first one is a data capture tool which takes advantage of the Intel PEBS (Precise Event Based Sampling) [69] to sample Flops and bytes transferred from main memory to the processor. This tool is an evolution of the tool presented in 2.2 and, in its current version, it obtains both memory accesses data and 3DyRM data. It uses the `perf_events` interface from the Linux kernel to interact with the HC, which makes it highly portable. The information obtained by this tool is processed by our second tool in an R environment [73] to generate the model and to visualise it. Using this second visualisation tool, the 3DyRM can be easily rendered for each core, thread or process in the system, or even displayed as a video, showing the evolution of the execution of the monitored programs.

Intel PEBS captures the entire content of the core registers in a buffer each time it detects a certain number of hardware events. These registers include hardware counters, which can be measuring other events. The data capture tool uses two PEBS buffers. One of them captures floating point information each time a certain number of instructions has been executed. This number can be fixed by the user, determining the instruction sampling rate. The other one captures the detailed information of a memory load event, including its latency, after certain number of memory load events, in the same way as it was explained in Section 2.2. The user not only can select the memory sampling rate, but the minimum load latency that an event must have in order to be counted, allowing the user to focus only on the loads he is interested in. Note that, the instruction sampling rate and the memory sampling rate can be different. These sampling rates determine the overhead (overhead will be detailed in Section 3.4.1).

To obtain the information needed to create the 3DyRM, the number of floating point operations executed by each core must be extracted. This means that at least ten different events must be considered in Intel Sandy/Ivy Bridge [28] processors. These events are in the set of FP_COMP_OPS_EXE: SSE_SCALAR_DOUBLE and FP_COMP_OPS_EXE: SSE_FP_SCALAR _SINGLE. As stated in Section 1.3, only 4 counters, counting 4 events, can be used at the same time without time multiplexing, which complicates counting all possible floating point operations. Anyway, if no packed floating point operations are considered, only two of these events can be taken into account: FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE and FP_COMP _OPS_EXE:SSE_FP_SCALAR_SINGLE. Additionally, data traffic between main memory modules and caches have to be considered for each core. Therefore, virtual addresses that produce cache misses have to be stored by using the OFFCORE_REQUEST: ALL_DATA _READ event. The sampling frequency is established through the number of instructions executed by each core. In this way, information about the number of instructions, the number of floating point operations, and the number of data read are stored.



/Volumes/Oscar3/t7600/Codigos/PEBS/Roofline/iprofile-multi_roof-m-11111111-m0

Plot: instructions_time_by_cpu_by_tid

CPUs: ◇ 0 ◇ 1 ◆ 2 ◇ 3 ◇ 4 ◇ 5 ◇ 6 ◇ 7

PID: 4084

TID: 4086

minGFlops: 0

maxGFlops: 2

minFlopsb: 0

maxFlopsb: 10

minLatency: -1

maxLatency: 2000

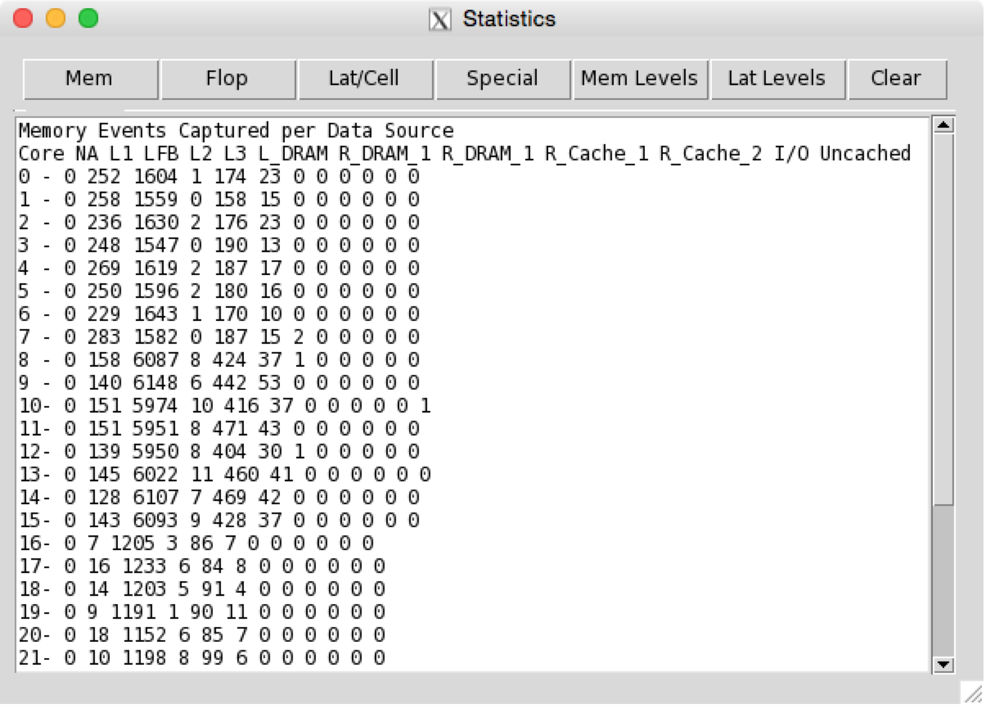
CPU	PID	TID	M_GFLOPS	M_FLOPSB	M_LATENCY	TIME
0	4084	4084	0.1481933	0.8895182	846.9125	310816796233
1	4084	4085	0.1551502	0.9280041	653.3619	309945031110
2	4084	4086	0.1550856	0.9264674	684.4544	310061703047
3	4084	4087	0.1555435	0.9355620	669.6800	309314067924
4	4084	4084	0.0000000	0.0000000	993.0500	13751874081
4	4084	4088	0.1476702	0.9288844	804.4058	333963577737
5	3359	3359	NA	NA	0.0000	102050823386
5	4084	4089	0.1469846	0.9218777	859.6415	333938622744
6	3327	3327	NA	NA	1099.0000	237369811446
6	4084	4090	0.1464475	0.9030489	843.2629	333925963624
7	4084	4091	0.1473392	0.9219820	800.6977	333885208154

Figure 3.5: The Graphical User Interface in R.

3.3.1 Performance visualisation tool

To deal with the study of the performance data we have used the R environment. A series of R functions were implemented to read and process the performance data obtained by the HC, and a Graphical User Interface (GUI) was implemented (see Figure 3.5). This interface allows to read and process the data, as well as to draw several graphs and figures to show relevant performance data. These figures range from simple graphics showing the evolution in time of different performance metrics (such as memory latency or instructions retired per second) to more complex ones, like the DyRM and 3DyRM models. It can even show the DyRM as an animation to highlight its evolution through time.

To obtain a complete image of the system, the performance tool can show the information in different ways. It is possible to focus on the hardware, and show figures for each system core individually, showing the performance without caring for which process is responsible. It is also possible to focus on processes or threads and represent models for each `pid` or `tid` in execution, even following them between core migrations. In this way, users may have a complete view of the system performance. This application can show composites made of different kinds of figures simultaneously. The tool can also show relevant statistics, such as mean GFLOPS, mean memory latency, or even the data source in cache misses (that is, the memory level were data was found after a cache miss), from different perspectives (core, thread, node,...). An example of a summary screen provided by the visualisation tool is shown in Figure 3.6. In this figure, the number of data accesses captured in each core from every possible data source in a 4 node system are shown, as given by the HC.



```
Memory Events Captured per Data Source
Core NA L1 LFB L2 L3 L_DRAM R_DRAM_1 R_DRAM_1 R_Cache_1 R_Cache_2 I/O Uncached
0 - 0 252 1604 1 174 23 0 0 0 0 0 0
1 - 0 258 1559 0 158 15 0 0 0 0 0 0
2 - 0 236 1630 2 176 23 0 0 0 0 0 0
3 - 0 248 1547 0 190 13 0 0 0 0 0 0
4 - 0 269 1619 2 187 17 0 0 0 0 0 0
5 - 0 250 1596 2 180 16 0 0 0 0 0 0
6 - 0 229 1643 1 170 10 0 0 0 0 0 0
7 - 0 283 1582 0 187 15 2 0 0 0 0 0
8 - 0 158 6087 8 424 37 1 0 0 0 0 0
9 - 0 140 6148 6 442 53 0 0 0 0 0 0
10- 0 151 5974 10 416 37 0 0 0 0 0 1
11- 0 151 5951 8 471 43 0 0 0 0 0 0
12- 0 139 5950 8 404 30 1 0 0 0 0 0
13- 0 145 6022 11 460 41 0 0 0 0 0 0
14- 0 128 6107 7 469 42 0 0 0 0 0 0
15- 0 143 6093 9 428 37 0 0 0 0 0 0
16- 0 7 1205 3 86 7 0 0 0 0 0 0
17- 0 16 1233 6 84 8 0 0 0 0 0 0
18- 0 14 1203 5 91 4 0 0 0 0 0 0
19- 0 9 1191 1 90 11 0 0 0 0 0 0
20- 0 18 1152 6 85 7 0 0 0 0 0 0
21- 0 10 1198 8 99 6 0 0 0 0 0 0
```

Figure 3.6: Statistics screen. Memory accesses captured by data source.

As an example of these functionalities, Figures 3.7, 3.8 and 3.9 show the execution of two NPB-OMP benchmarks [30] run one after the other, first the `ep.A` and then the `ft.A`, in our Xeon Server X (see Section 1.2.2). In Figure 3.7, the complete evolution of the performance is shown, using a DyRM for each core. In this figure, each point in the DyRM is coloured relative to the total data capture time. In Figure 3.8, the DyRM for core 2 and the applications there executed are shown separately. Note that the DyRM colouring is made relative to the total time, and the tool itself scales data to obtain an accurate vision of each thread performance evolution. Finally, in Figure 3.9, the 3DyRM is show, with performance information for the whole system, and all cores combined in it. Here colouring depends on the processor (temporal colouring could also be used), so each processor data has a different colour to highlight possible differences in performance between them (such as memory latency due to remote RAM accesses). In this particular case, there are no visible performance differences between processors but differences between both benchmarks can be appreciated. Note that the information shown in Figure 3.8 is the same as in Figure 3.9(a).

3.4 Case studies

In this section, performance results for some of the NPB3.3-OMP benchmarks [30] are shown, executed on two different systems, Xeon Servers X and Y (see 1.2.2 for more details). All executions on these systems were carried out with 16 threads, disabling multithreading. The NPB benchmarks we considered in this study are: CG (Conjugate Gradient), FT (Discrete 3D Fast Fourier Transform), EP (Embarrasingly Parallel), and the solvers LU (Lower-Upper Gauss-Seidel), BT (Block Tri-diagonal), and SP (Scalar Pentadiagonal).

In the next subsections the overhead of the data capture tool and some results are discussed. In particular, the effects of diverse changes in the benchmark codes studied are explained using the DyRM and 3DyRM models.

3.4.1 Overhead of data capture

As previously stated (in subsection 1.3.2), the overhead from using PEBS is determined by the sampling rates. Obtaining more detailed information requires a higher sampling rate and a

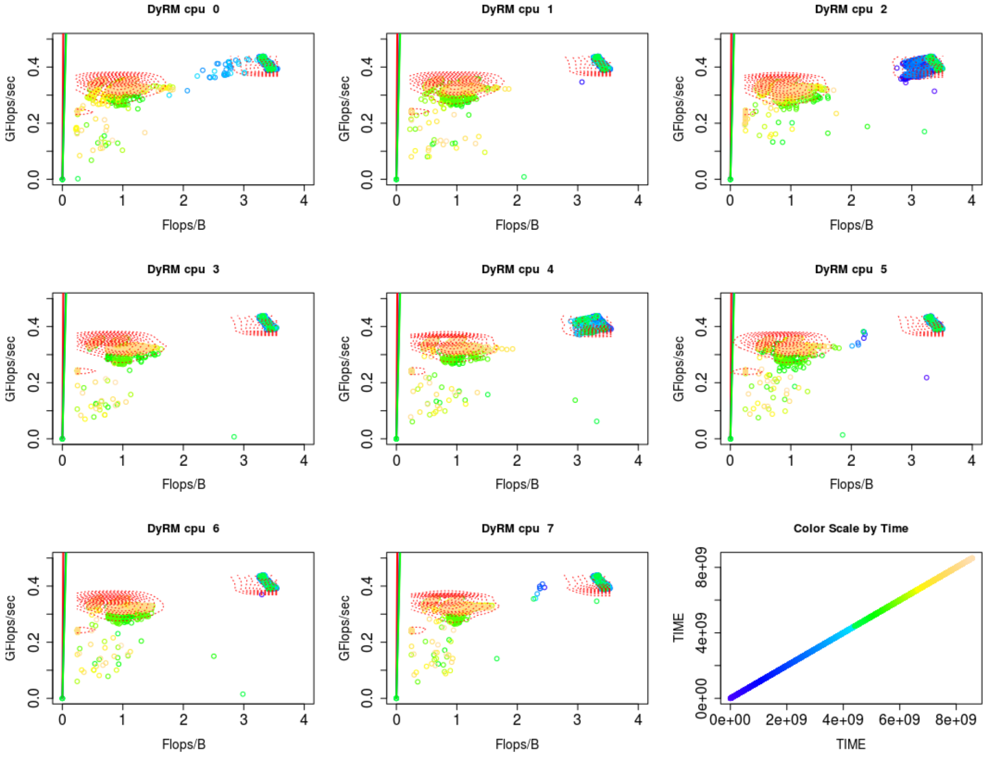
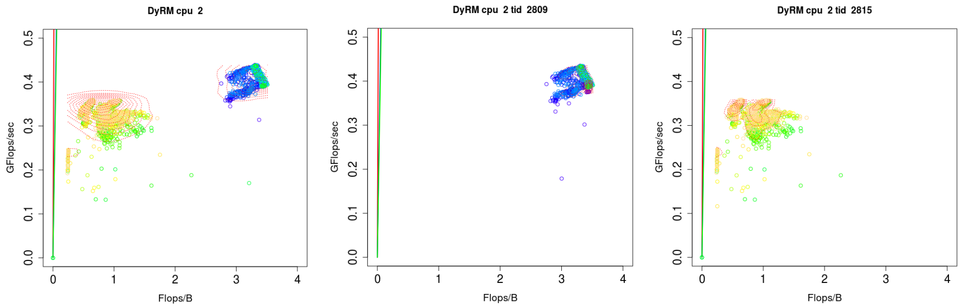


Figure 3.7: DyRM models for the 8 system cores.

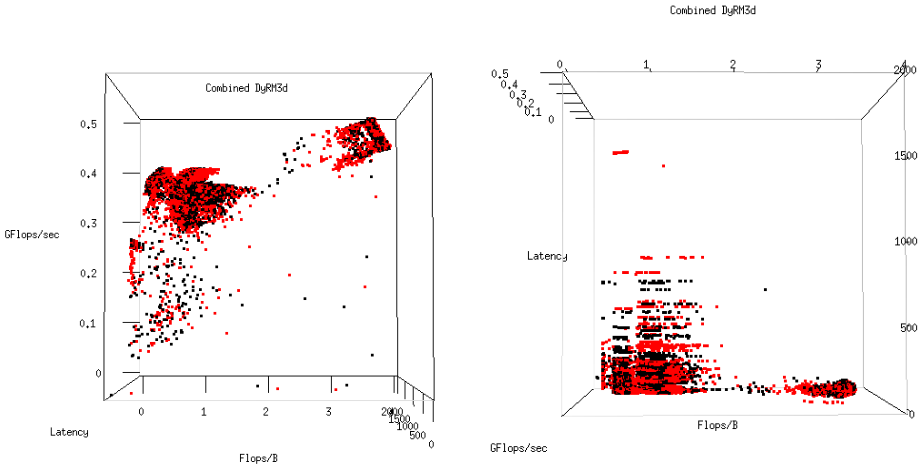
larger overhead. Since we want to sample both memory events and floating point information, there are two sampling rates. The 3DyRM is based on floating point performance, so each point in the model corresponds to a sampled event. As such, the more often floating point information is sampled, the more points per second the 3DyRM can render. The memory latency assigned to that point in the model is given by the mean latency of memory events captured in the previous time interval. So, if the memory events are captured in a rate close to that of the floating point information, each point is a close approximation of the actual latency in that time interval.

The overhead in this study on Xeon Server Y, executing each benchmark with 16 threads, is shown in Table 3.1. Two sets of sampling rates were used to illustrate the effect of the overhead. In this table the execution times for the NPB benchmarks studied, compiled with `O2`



(a) DyRM core 2 with ep.A and ft.A (b) DyRM core 2 tid 2809, ep.A (c) DyRM core2 tid 2815, ft.A

Figure 3.8: DyRM of ep.A y ft.A. Bechmark detection.



(a) 3DyRM, GFLOPS FlopsB (b) 3DyRM, FlopsB Latency(cycles)

Figure 3.9: 3DyRM de ep.A y ft.A. Processor 0 (cores 0-3) in red, processor 1 (cores 4-7) in black.

Table 3.1: Data capture overhead relative to the number of samples taken per thread. Ms/th/s, number of **m**emory operations **s**ampled per thread per second. Is/th/s, number of **s**amples of **i**nstructions counts per thread per second.

		Low Sampling Rate			High Sampling Rate		
Code	Time(s)	Ms/th/s	Is/th/s	Over.(%)	Ms/th/s	Is/th/s	Over.(%)
cg.A	0.27	45	23	**9.6**	224	118	**14.7**
ft.A	0.86	34	27	**4.1**	174	137	**5.2**
ep.A	1.58	23	27	**0.8**	116	133	**1.9**
lu.A	5.09	54	37	**2.4**	122	185	**4.1**
bt.A	5.73	47	57	**0.7**	176	278	**3.8**
sp.A	4.57	41	40	**1.2**	175	201	**3.5**
cg.B	10.60	41	21	**0.6**	322	259	**2.9**
ft.B	6.69	28	45	**1.1**	369	382	**4.1**
ep.B	5.33	21	37	**0.1**	208	316	**2.4**
lu.B	23.84	41	39	**0.5**	201	332	**3.9**
bt.B	28.53	31	55	**0.4**	303	326	**3.5**
sp.B	29.38	20	31	**1.0**	203	267	**3.8**

optimisation, are shown in column 2 (called Time). The next three columns show the mean number of memory events sampled per thread each second, the mean number of instruction counts sampled per thread per second, and the overhead incurred during measurement using low sampling rates. As shown in this table, to obtain a good resolution in the model, with about 40 points per second, overheads are not usually greater than 1%. Only benchmarks CG and FT present high overheads due to their small execution time, which means that the initialisation and execution of the data capture is important. We have got a significant overhead (about 3%) only when a high resolution is used, i.e. about 200 points per second, as shown in the last three columns of Table 3.1. All figures in this thesis were obtained using a resolution of approximately 40 points per second.

3.4.2 Floating Point overcounting

As it was described in Section 1.3.2, there is an issue with the floating point operations (FP_OPs) hardware monitoring in Intel architectures. In the Intel Sandy Bridge architecture, floating point operations counters count executed operations, not retired operations [86]. A FP_OP is will be reissued until its operands appear in the registers. This distorts the 3DyRM for low values of FlopsB when main memory is accessed aggressively.

To illustrate this problem, a test program was implemented and executed on the Xeon Server Y. This program simply computes the dot product of two vectors of float numbers (SDOT), and allows a stride t to be specified, which means only the values every t vector positions are multiplied. This stride is used to modify the operational intensity of the application, since data is brought from memory in cache lines of 64 bytes, that is, 8 floats at a time. From $t = 1$ to $t = 8$ the same data is brought to the cache, although fewer operations are executed every time. In a simple program like this, the theoretic number of floating point operations can be easily calculated and compared with the result provided by the hardware counters. The program was executed both placing all of the data in node 0 and sharing the data between nodes 0 and 1, with each core accessing to its local memory. Results of the flop overcounting, in percentage, are shown in Figure 3.10. Note that a low OI, with long stalls waiting for data, produces an overcount in the number of flops. Furthermore, when each core is operating with its local memory (the case with data in nodes 0 and 1) the overcounting is similar for all cores, since they all suffer the same latency, but when some cores access remote memory and others local (the case with all data on node 0) the overcounting varies greatly, due to the longer latency of some cores (for instance, for $t = 8$ the mean latency of accesses larger than 400 cycles was 910 cycles for node 0 and 1280 cycles for node 1).

As it can be seen in Figure 3.10, the overcounting can reach more than 700%. This creates problems in some cases for obtaining the 3DyRM. For instance, Figures 3.11(a) and 3.11(b) show the 3DyRM for this SDOT with $t = 8$ and all data on node 0. In this execution, cores on node 0 have lower latency and execute their work faster than those on node 1; once node 0 finishes its work, node 1 works alone until the end. Node 0 should show greater GFLOPS than node 1, except at the end, when node 1 is left alone and no longer has to compete with node 0 to access the memory. Nevertheless, in Figure 3.11(a) node 0 (in green) seems to

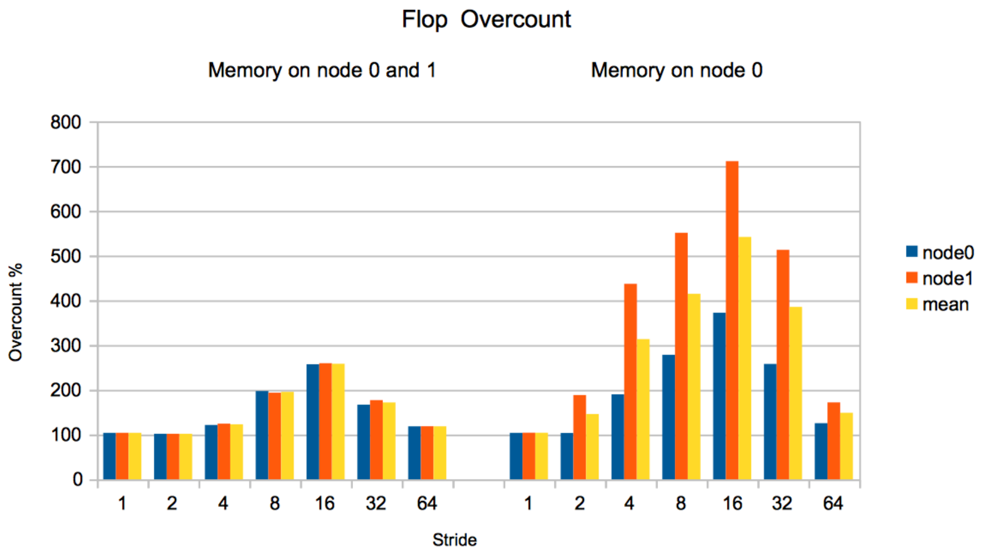


Figure 3.10: Flop overcounting results.

have lower GFLOPS than node 1 (in black) all the time. According to Figure 3.11(b), it seems that the node with the longer latency executes better. Even more, in Figure 3.11(a) it seems that there are three phases with different OI, which is not likely the case, since the application always performs the same operation. All these negative effects are due to flop overcounting. Summarising, as node 1 has larger latencies, it overcounts more flops than node 0, as it appears to present more GFLOPS and messes the FlopB count. This issue is illustrated and discussed in several forums in internet [86, 67, 29].

In Figures 3.11(c) and 3.11(d) an alternative model, similar to the 3DyRM, is shown. Although the use of number of instructions could have many drawbacks to measure performance, this new model uses data from instruction retired counts instead of floating point operations. So, GFLOPS are substituted by Giga Instructions Per Second (GIPS) and FlopsB by Instructions per Byte (InstB). This model shows, in Figure 3.11(c), how the equivalent to the OI measured with instructions does not change, because the operation is the same, and how node 0 has greater GIPS, except for some points, which are due to the time at the end when node 1 is running alone. In Figure 3.11(d), it is clear how a greater latency leads to a lesser performance, and clearly shows the two phases of node 1. This alternative model using instructions is called i3DyRM.

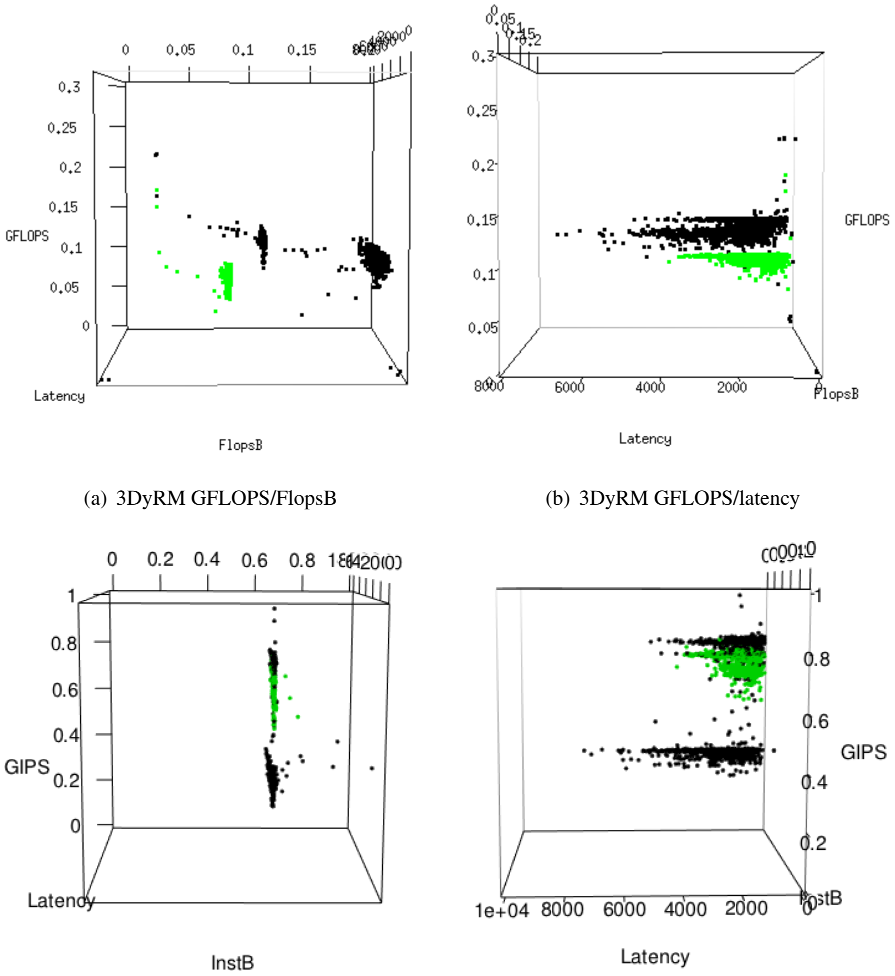
We consider the i3DyRM model complementary to the 3DyRM, and can be used in cases of low OI. Alternatively, this model can be obtained in systems with no hardware counters for floating point operations, like the Intel Haswell architecture [28], successor to Intel Ivy Bridge. Nevertheless, in the scope of the following subsections, the DyRM and 3DyRM models are sufficient, since used NPB-OMP benchmarks do not present an excessive floating point overcounting.

3.4.3 Effect of compiler optimisations

To illustrate the use of the tool and the utility of the model, the effect of general optimisations in a code was analysed, throughout the analysis of the behaviour of a NPB benchmark compiled without optimisation and with an O2 optimisation level. The DyRM of a FT benchmark is shown in Figure 3.12, executed on System X with 8 threads. Note that different phases can be identified using the information provided by the model. Optimising the program improves the GFLOPS count, but the difference among the phases persists, which may indicate the need to optimise each one separately.

3.4.4 Effect of the problem size

In this section we show the effects in our model of different problem sizes in the NAS benchmarks, executed on the Xeon Server X with 8 threads. Only the most representative benchmarks are shown. For example, the CG benchmark presents an initialisation phase, which shows a low performance behaviour. This is shown in Figure 3.13 in the lower left corner of each graph, with a blue colour, which means it happens early in time. This figure shows only the information for one core, but all cores show a similar behaviour. Note that this is a memory bound benchmark, and, as problem size increases, both OI and GFLOPS count decrease, while the initialisation phase still remains important. It hits the slanted part of the roof, since it is memory bound. In the case of size C, the performance overcomes the second roof, although not the top one. It means that it achieves a better bandwidth than expected, but obviously not more than the limit imposed by hardware.



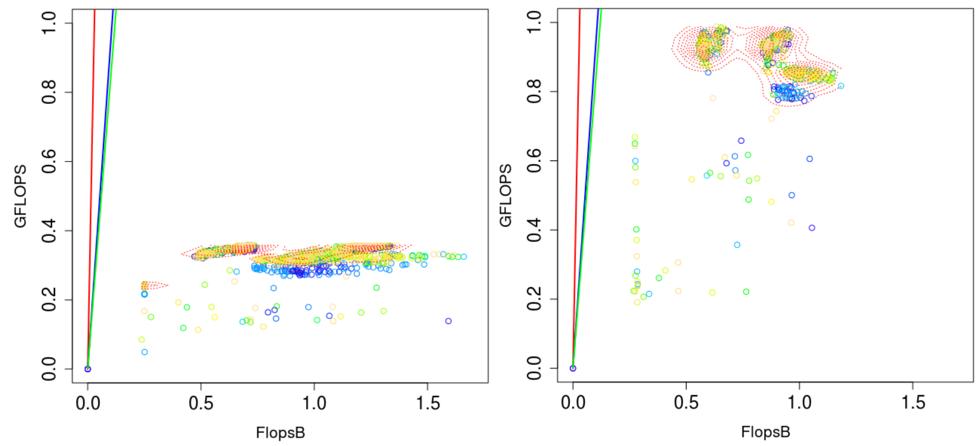
(a) 3DyRM GFLOPS/FlopsB

(b) 3DyRM GFLOPS/latency

(c) i3DyRM GIPS/InstB

(d) i3DyRM GIPS/latency

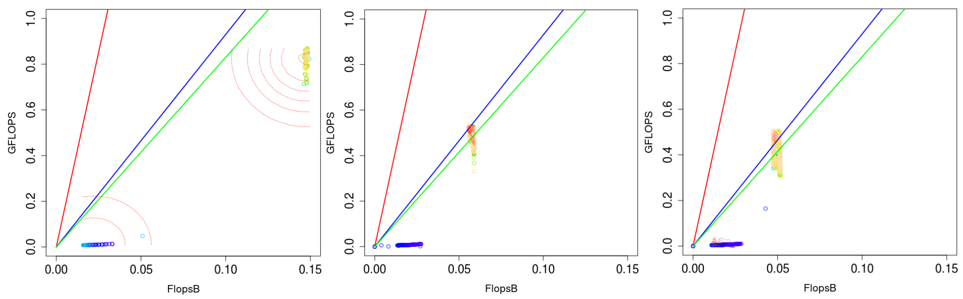
Figure 3.11: 3DyRM and i3DyRM for an SDOT with $t = 8$, node 0 in green, node 1 in black.



(a) No Optimisations

(b) O2 optimised

Figure 3.12: DyRM for FT.A on Xeon Server X (core 0).

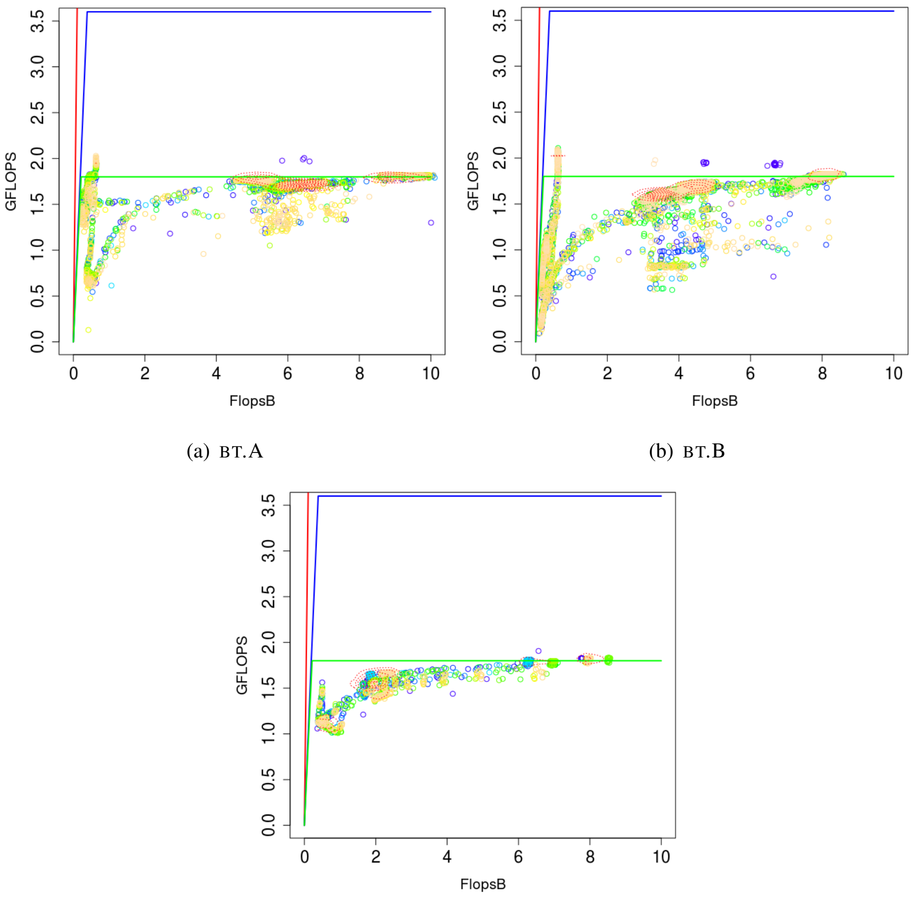


(a) CG.A

(b) CG.B

(c) CG.C

Figure 3.13: Roofline for CG (Sizes A, B and C) on Xeon Server X.



(a) BT.A

(b) BT.B

(c) BT.C

Figure 3.14: DyRM for BT benchmark (Sizes A, B and C) on Xeon Server X.

For the BT benchmark (Figure 3.14) at least three phases were found. Note that, as the problem size increases, the GFLOPS count stabilises, but this is not the case for the OI, which decreases. Anyway, this is clearly a compute bound program, and the use of SIMD instructions is advisable.

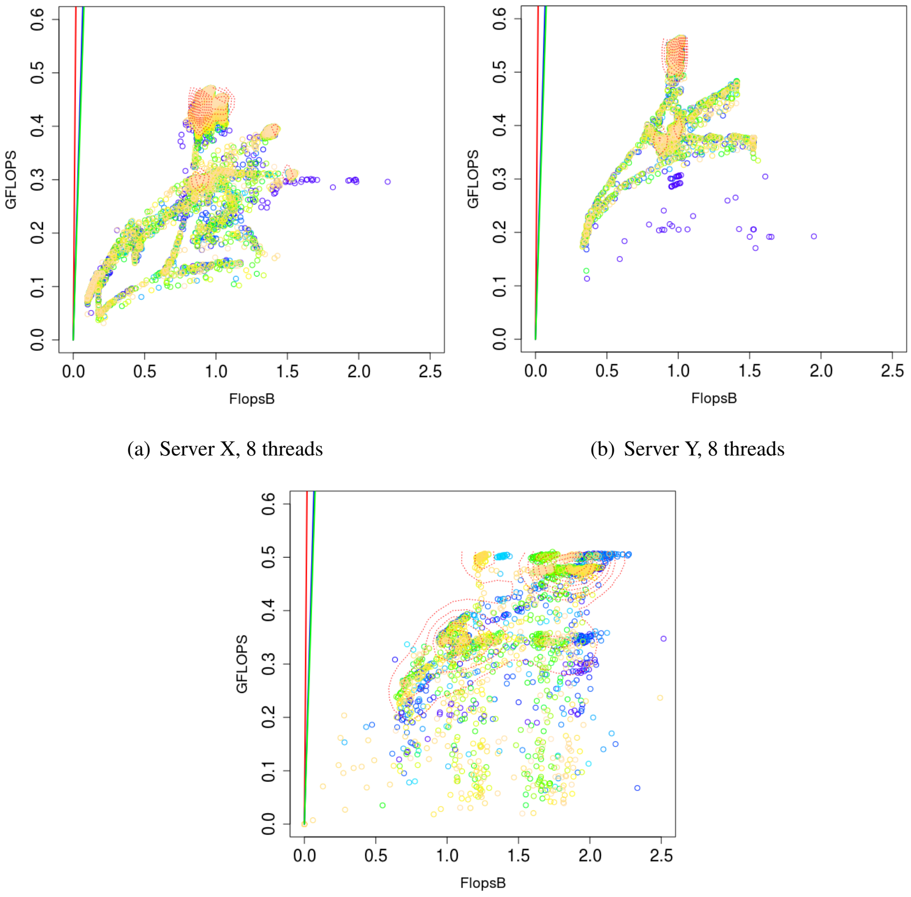
3.4.5 Comparison among processors

Figure 3.15 shows the DyRM in one core of a LU.A benchmark executed on two different processors, with different number of threads. Figure 3.15(a) shows the benchmark results on Xeon Server X, whereas in Figures 3.15(b) and 3.15(c) the execution takes place on System Y. The difference in GFLOPS between both processors is clear. Figure 3.15(b) shows the results when the benchmark is executed with 8 threads, while in Figure 3.15(c) it is executed with 16 threads. Due to the power scaling features of the Xeon E5-2650L, Figure 3.15(c) does not attain a GFLOPS count as high as the one in Figure 3.15(b).

3.4.6 The effect of latency

On a NUMA system each core may be at a different latency from different cells of main memory. That is, it may have a different affinity to various modules of main memory. Interconnexion between cores and memory may also vary inside the same shared memory machine. These differences may affect performance, and can be modelled using the main memory access time from each core.

Figure 3.16 shows the representation of the 3DyRM model with the EP.B benchmark on the Xeon Server X. Note that one processor achieves better OI than the other (Figure 3.16(a)). We can see clearly how, after an initialisation period, processor 0 executes the benchmark more efficiently than processor 1. Figure 3.16(b) shows that the accesses with longer latencies correspond to processor 1. The reason is that, in this case, in the EP.B benchmark, data is initialised by only one thread, on processor 0. This means it is stored in the nearest memory cell. Afterwards the benchmark launches all the other threads. Threads in processor 1 have a less efficient access to the data, because they are in the memory cell nearest to processor 0, leading to higher latencies and lower OI and GFLOPS.

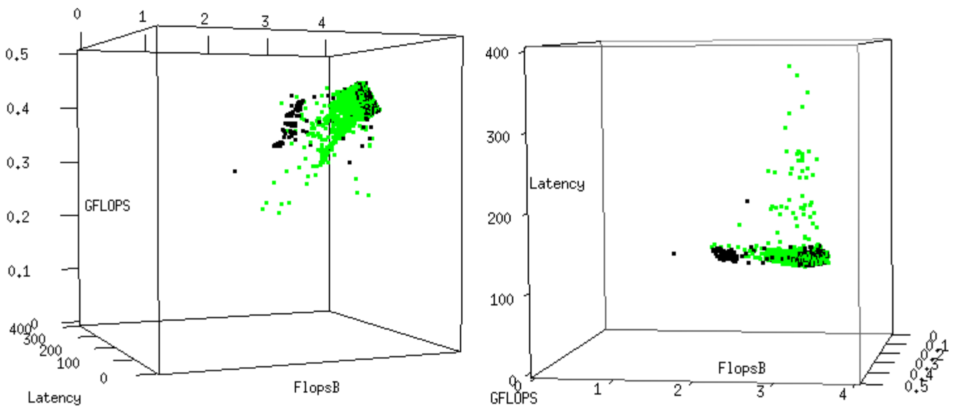


(a) Server X, 8 threads

(b) Server Y, 8 threads

(c) Server Y, 16 threads

Figure 3.15: DyRM for LU.A on different systems.



(a) GFLOPS/FlopB

(b) Latency(cycles)/FlopB

Figure 3.16: 3DyRM of EP.B in Xeon Server X with 16 threads. Data from processor 0 is shown in black, those from processor 1 is in green.

3.5 Recap

In Section 3.1 the Berkeley Roofline Model was introduced. While it remains a useful and simple model, it hides some characteristics of the applications that become important in many systems, specially manycore, NUMA or heterogeneous systems.

To overcome these limitations, a set of extensions to the Berkeley Roofline Model have been proposed in Section 3.2. The first extension shows the evolution of a program during its execution, and the second one shows, additionally, the differences of memory access latency among threads. To obtain the data for these models, advantage was taken of the PEBS counters of Intel processors.

A set of tools to automate the task was also implemented, allowing us to gather useful information with low overhead, they were presented in Section 3.3. We have shown their usefulness with a set of cases on two different multicore systems in Section 3.4, were their overhead and issues with floating point counting were detailed.

We use the proposed tools to show how parallel applications like the NPB-OMP benchmarks present complex behaviours and imbalances on multicore systems. These problems can be easily modelled with our tools, without influencing the normal execution of the applications, and showing a realistic model of their performance. Thanks to the proposed extensions of the Roofline Model, a program’s behaviour, its phases or imbalances can be more easily detected, making it easier to deal with performance issues.

Manuals for the applications presented in this chapter, as well as the source code of the tools, can be found at the CiTIUS git repository [62].

Chapter 4

Thread migration based on runtime information

As stated previously, multicore systems present on-board memory hierarchies and communication networks that influence performance when executing shared memory parallel codes. Characterising this influence is complex, and understanding the effect of particular hardware configurations on different codes is of paramount importance. In preceding chapters, monitoring information extracted from hardware counters at runtime has been used to characterise the behaviour of each thread in the parallel code in terms of the number of floating point operations per second, operational intensity, and latency of memory access. In this chapter, we propose to use this information to guide thread migration strategies that improve execution efficiency by increasing locality and affinity.

4.1 Introduction

Current microprocessors implement multicores that feature a diverse set of compute cores and on board memory hierarchies connected by increasingly complex communication networks and protocols with area, energy, and performance implications. For a parallel code to be correctly and efficiently executed in a multicore system, it must be carefully programmed, and memory sharing stands out as a sine qua non for general purpose programming [80]. A critical programming challenge for these systems is to partition application tasks, mapping them to one of many possible core thread configurations to achieve a desired performance in terms of throughput, delay, power, and resource consumption, among others [31]. The number of mapping choices increases as the number of cores and threads increase.

Considering the architectural features, particularly those that determine the behaviour of memory access, it is critical to improve locality of access and affinity among threads, data, and cores. Both locality and affinity are important to reduce the access latency to data. In addition, a large fraction of the on-chip multicore interconnect traffic is originated not from actual data transfers but from communication between cores to maintain data coherence [75]. An important impact of this overhead is the on-chip interconnect power and energy consumption.

Moving threads close to where their data reside can help alleviate memory related performance issues, since when threads migrate, the corresponding data usually stays in the original memory module, and is accessed remotely by the migrated thread. This could induce inefficiency that, sometimes, cannot be alleviated by the benefits of the migration [11, 78, 79, 38, 33]. Alternatively, memory pages can be migrated, instead of threads, to improve performance [2]. Some results and studies are available for multicore processor analysis. For example, [21] performed a mean value analysis of a multithreaded multicore processor and showed that there is a performance decrease to be avoided as the number of threads increases. Markovian models were used in [8] to model a cache memory subsystem with multithreading, and other works [4, 31] have modelled multithreaded multicore using queuing theory. Proposals for heterogeneous multicore [34] move threads between cores to exploit power-performance trade-offs. Performance information can be used to guide thread migration strategies to improve the efficiency of the execution of the code by increasing data locality and/or thread affinity. Each such migration incurs overhead, similar to a context switch [37, 39], from saving and restoring processor states and virtual machine control structure, extra translation lookaside buffer misses and related page walks, cache misses, and interrupt rerouting. The indirect overhead of TLB and cache misses for a migration is potentially higher than for a context switch, because the migrated thread begins execution in a different processor environment and cache hierarchy. The performance benefits of saving and restoring cached data during migration are analysed in [74]. Thread startup performance can be accelerated after migration by predicting and prefetching the working set of the application into the new cache according to [6].

In the previous chapter various performance models have been commented [58, 1, 59, 16, 10, 57, 76]. In particular, the Roofline Model (RM) [85] was detailed. Based on this model the 3DyRM model was proposed.

In this chapter, we propose using the 3DyRM to implement strategies for migrating threads in shared memory systems [48]. In particular, based on the parameters of this model, new functions that characterise the efficiency of each thread are proposed. As a result, just a single value is used to quantify the behaviour of each thread in terms of locality and affinity.

4.2 Migration strategies and algorithms

In this thesis, a tool to monitor a program execution and to perform thread migrations was implemented. Given a target parallel shared memory program to be optimised, this tool captures information about the behaviour of every thread on the system. The migration tool is constantly gathering performance data in terms of the parameters that define the 3DyRM, that is, `GFLOPS`, `flopsB` and `latency` for each core and thread. This information is proposed to be used to guide a thread migration mechanism that is activated every T miliseconds. After each migration, performance data must be initialised, since the configuration has changed.

The concept is to use the defining parameters of 3DyRM as objective functions to be optimised [50]. Thus, it can be considered as a multiobjective optimisation (MOO) problem. The proposed technique is an iterative method inspired in evolutive optimisation algorithms. To this end, we define an utility function to represent the relative importance of each of the 3DyRM parameters. This function is a weighted product that can be considered as representative of the performance of each parallel thread, and the parameters characterise the efficiency of each thread. Thus, a single value is proposed to quantify the performance of each thread in terms of locality and affinity. A number of methods for the MOO problem can be found in the literature [52]. The aim of many of them is to obtain Pareto optimality numerically, but this task is usually computationally complex and different approaches were proposed.

Note that, in our case, there are no functions to be optimised, but a set of values that are measured in the system. Therefore, we propose to apply multiobjective optimisation methods to deal with our problem by using the values of these parameters measured on the fly. Thread migration is then used to modify the state of each thread trying to simultaneously optimise these three target parameters.

In following sections, three proposals for thread migration strategies, based on three migration algorithms, are presented. They are, in increasing complexity, the Interchange Migration Algorithm (IMA) [47, 49], the Interchange Migration Algorithm with performance Record (IMAR), and the Interchange Migration Algorithm with performance Record and Rollback (IMAR^2) [63]. For all these algorithm case studies were made to show their usefulness.

4.2.1 IMA Interchange Migration Algorithm

`GFLOPS`, `flopsB` and `latency` (the 3DyRM parameters) are considered as optimisation functions whose values present different orders of magnitude. For this kind of situations, to aggregate these parameters, the use of weighted product methods is recomended [9], characterising each thread by the value of an aggregated objective function P that combines the three parameters. In particular, we propose the following function for the i-th thread, where M is the number of threads:

$$P_i = \frac{\text{GFLOPS}^{\beta} \cdot \text{flopsB}^{\gamma}}{\text{latency}^{\alpha}} \qquad i = 1, \ldots, M \tag{4.1}$$

Each thread is ranked with its corresponding P_i, where higher values of P_i indicate better performance[1]. Note that the importance of each factor is scaled by a different weighting factor α, β and γ, which indicates the relative significance of the corresponding objective function. Therefore, this function is the result of scalarising the multiobjective optimisation problem [52] as the product of the three objective values weighted by these three factors. This

[1] Because of the implementation of the migration tool, placing the `latency` value in the lower term of the division gave problems with the floating point operations; so the actual tool uses the inverse of the P function. The algorithm is functionally the same.

approach was successfully used in different applications [9, 7, 83]. Note that this function produces a value whose dimensionality is not related with any performance metric.

As discussed in section 3.4.2, `GIPS` and `instB` may be substituted for `GFLOPS` and `flopsB`, where appropriate, using an alternate form of equation 4.1,

$$P_i = \frac{\text{GIPS}^{\beta} \cdot \text{instB}^{\gamma}}{\text{latency}^{\alpha}} \qquad i = 1, \ldots, M \tag{4.2}$$

During the execution of the code to be optimised, values of P_i are computed each T ms for each thread. Based on these values, a certain number $\Theta < M$ of threads are selected to be migrated. This process is inspired in iterative evolutive optimisation methods, in which random selection plays a key role, in particular to improve convergence. Different strategies can be considered to select threads to be migrated. For the IMA algorithm, two strategies are proposed:

1. W_RBEST, to swap the $\Theta/2$ threads that present the highest values of P with $\Theta/2$ threads selected randomly among the rest of them.

2. W_RWORST, to swap the $\Theta/2$ threads that present the lowest values of P with $\Theta/2$ threads selected randomly among the rest of them.

To avoid ping-pong effects in the migration of any thread, both version of the IMA avoid migrating the same threads two consecutive times. Note that these strategies also contribute to balance the workload among threads. To simplify notation, an IMA including its parameters is denoted as IMA$[T;\Theta;\alpha,\beta,\gamma]$.

4.2.2 IMAR Interchange Migration Algorithm with performance Record

As an alternate proposal, the initial migration algorithm presented in the previous section was modified to make it more predictive and to take into account past performance. This new algorithm is called Interchange Migration Algorithm with performance Record (IMAR).

This algorithm functions the same way as the IMA, but adds a performance record. Instead of having just one instant value P of performance for each thread, the IMAR keeps a record with the previous performance of each thread in each node of the system.

Let P_{ij} be the performance for the i-th of M threads on the j-th of N nodes. Then, for each iteration of the aggregate function,

$$P_{ij} = \frac{\text{GFLOPS}_{ij}^{\beta} \cdot \text{flopsB}_{ij}^{\gamma}}{\text{latency}_{ij}^{\alpha}} \qquad i = 1,\ldots,M \quad j = 0,\ldots,N-1 \tag{4.3}$$

where $\text{GFLOPS}_{ij}^{\beta}$ is the `GFLOPS` of the thread scaled by β, $\text{flopsB}_{ij}^{\gamma}$ and $\text{latency}_{ij}^{\alpha}$, are the `flopsB` and latency values scaled by γ and α, respectively, and larger values of P_{ij} imply better performance.

The alternate form of equation 4.3 using instruction counts would be:

$$P_{ij} = \frac{\text{GIPS}_{ij}^{\beta} \cdot \text{intsB}_{ij}^{\gamma}}{\text{latency}_{ij}^{\alpha}} \qquad i = 1,\ldots,M \quad j = 0,\ldots,N-1 \tag{4.4}$$

Initially, no values of P_{ij} are available for any thread on any node. For each iteration, P_{in} is computed for every thread in the system and stored, where n is the node where the i-th thread is being executed at that moment. If there is a previous value of P_{in}, the new value replaces the previously saved one. Thus, the algorithm adapts to possible behaviour changes for the threads. For example, in a Xeon server with four nodes, $N = 4$, four values of P (one for each thread) are saved each iteration. As threads migrate and are executed on different nodes, matrix P_{ij} is progressively updated and filled.

Unlike IMA, IMAR does not allow to select the number of threads to be moved per iteration, instead the value of T is used to increase or decrease the number of migrations per unit of time. At each iteration, every T milliseconds, once the new values of P_{ij} are computed, the thread with the worst current performance, in terms of P_{ij}, is selected to be migrated. To compare threads from different processes, each individual P_{in} is normalised by dividing it by the mean of all threads of the same process, identified by its PID,

$$\widehat{P_{ic}} = \frac{P_{ic}}{\overline{P}_{jc}} \qquad \forall j/\text{PID}(i) = \text{PID}(j) \tag{4.5}$$

where $\overline{P}_{jc}$ is the mean performance of all the threads in the same process as i. Thus, for each process, those threads with $\widehat{P_{in}} < 1$ are currently performing worse than the mean of the threads in the same process, and the worst performing thread in the system is considered to be the one with the lowest $\widehat{P_{in}}$, i.e., the thread performing worse when compared to the other threads of its process. This is the migration thread, denoted by Θ_m.

The migration can be to any core in a node other than n. A weighted random process is used to choose the destination, based on the stored performance values. The aim is to consider all possible migrations, and so all P_{ij} values are updated and behavioural changes are incorporated. To ensure the migration will globally improve performance, every possible destination is granted a number of tickets according to the likelihood of that migration improving performance, and the destination with the larger likelihood overall is chosen. Migration may take place to an empty core, where no other thread is currently being executed, or to a core occupied with other threads. If there are already threads in the core, one would have to be swapped with Θ_m. The swap thread is denoted as Θ_g, and all threads are candidates to be Θ_g. While not all threads may be selected to be Θ_m, e.g. a process with a single thread would always have $\widehat{P_{in}} = 1$ and so never be selected, they may still be considered for Θ_g to ensure we obtain the best possible performance for the system as a whole.

The rules applied to distribute tickets (B) for the random selection procedure are:

– Destinations in nodes where Θ_m has previously performed worse than the current node get B_1 tickets.

– Destinations in nodes where there is no previous data recorded for Θ_m get B_2 tickets.

– Destinations in nodes where Θ_m has previously performed better than the current node get B_3 tickets.

The best migration should be that which results in good performance from both threads, Θ_m and Θ_g. Therefore, additional tickets are awarded to each destination according to the values of P_{gn}, where g is Θ_g, and n is the node that currently hosts Θ_m:

– Destinations where Θ_g has previously performed worse in n in the past get B_4 tickets.

– Destinations with no previous information for Θ_g get B_5 tickets.

– Destinations where Θ_g has previously performed better get B_6 tickets.

– Destinations for cores with no threads assigned get B_7 tickets.

Although P_{ij} are only saved for nodes, by including the performance of the possible Θ_g, different cores in the same node, and even different threads in the same core, may get a different number of tickets.

Suitable choice of B_k is critical, and this is discussed further below. When all tickets have been assigned, a final destination core is randomly selected based on the awarded tickets. The interchanging thread, Θ_g, is chosen from those currently being executed on that core, if the core is not free. Once the threads to be migrated are selected, the migrations are actually performed.

To simplify notation, an IMAR including its parameters is denoted as IMAR$[T;\alpha,\beta,\gamma]$.

4.2.3 IMAR^2 Interchange Migration Algorithm with performance Record Rollback

After considering the results of the IMAR algorithm some changes were proposed to improve the migration algorithm. Note that, migrations may affect not only the involved threads, Θ_m and Θ_g, but all threads in the system due to synchronisation or other side effects among threads. These relations are not accurately modelled using each thread performance separately. Therefore, we propose the interchange algorithm with performance record and rollback (IMAR^2), where the total performance for each iteration is calculated as the sum of all P_{ic} for all threads. Thus, the current total performance, $Pt_{current}$, that is, a single value, is available to evaluate a thread configuration, independent of the particular processes being executed. The total performance of the previous iteration is denoted as Pt_{last}. An acceptable ratio, $0.9 < \omega \leq 1$ is previously defined for $Pt_{current}/Pt_{last}$.

Incorporating these concepts, decisions are made regarding the next iterations of the algorithm. The algorithm may dynamically adjust the rate of migrations by changing T between a given minimum, T_{min}, and maximum, T_{max}, doubling or halving the previous value. A rollback mechanism was also implemented, to undo migrations if they result in a significant loss of performance, returning migrated threads to their former locations. If a rollback is performed, no other migrations are made during that iteration.

Thus, the rules guiding the procedure are:

- If $Pt_{current} \geq \omega Pt_{last}$, i.e., the total performance remains stable or improves: Migrations are considered productive, T is halved ($T \rightarrow T/2$), and a new migration is performed according to IMAR.

- If $Pt_{current} < \omega Pt_{last}$, i.e., the total performance decreases more than a given threshold: Migrations are considered counter-productive, T is doubled ($T \rightarrow 2 \times T$), and the last migration is rolled back.

IMAR^2 considers that, on one hand, if a thread placement has low total performance, migrations should be performed to obtain better thread placement. In this case, migrations are likely to increase performance ($Pt_{current} \geq \omega Pt_{last}$) so T is decreased to perform migrations more often and reach optimal placement quicker. On the other hand, if thread placement has high total performance, migrations have a greater chance of being detrimental. In this case, if $Pt_{current} < \omega Pt_{last}$, there is no requirement for many migrations, so T is increased. The algorithm continues to migrate threads to allow for changes in system behaviour, and to obtain performance information, rolling these back if necessary. To simplify notation, IMAR^2 with its parameters is denoted as IMAR$^2[T_{min}, T_{max}; \alpha, \beta, \gamma; \omega]$.

4.2.4 IMAR example

A simple example is presented to clarify our proposal. Consider a system with 6 cores in three different nodes, incorporating three processes, each with two threads. Initially, Process 1 has threads 100 and 101 executed in node 0 (cores 0 and 1), Process 2 has threads 200 and 201 executed in node 1 (cores 2 and 3), and Process 3 has threads 300 and 301 executed in node 2 (cores 4 and 5), as shown in Table 4.0(a), where threads are shown with the core they currently reside, and their recorded performance in each node. Nodes where threads have not been executed previously have no performance information recorded.

Table 4.0(b) shows a later state, where some migrations have been executed and more performance information is available. The performance of each thread in its current node is shown in bold. Suppose a migration has to be decided at this point. Table 4.2 shows each thread's performance and normalised performance $\widehat{P_{in}}$ (equation 4.5). In this example thread 300 has the worst relative performance, so Θ_m=300.

(a) Initial state.

thread (core)	P_{i0}	P_{i1}	P_{i2}
100 (0)	**2.4**	–	–
101 (1)	**2.6**	–	–
200 (2)	–	**1.4**	–
201 (3)	–	**1.6**	–
300 (4)	–	–	**6.3**
301 (5)	–	–	**5.2**

(b) State after *i*-th iteration

thread (core)	P_{i0}	P_{i1}	P_{i2}
100 (2)	2.5	**1.9**	2.9
101 (4)	2.7	1.8	**3.1**
200 (0)	**0.9**	1.4	–
201 (5)	–	1.6	**2.1**
300 (1)	**3.3**	–	6.3
301 (3)	–	**8.1**	5.7

Table 4.1: Example of use of IMAR. Thread state.

The case studies, Section 4.5, show good values for B_k to be the following:

- $B_1 = B_4 = 1$: previous low performances are penalised,
- $B_2 = B_5 = 2$: allow more performance information to be obtained,
- $B_3 = B_6 = 4$: previous good performances are rewarded, and
- $B_7 = 3$: allow migrations to free cores and improve load balance.

With these values, a thread interchange that would increase the performance of both threads involved would get eight tickets, the maximum, whereas one that would worsen the performance of both threads would get only two tickets, the minimum. Migrations and interchanges where there are no data still have a chance of being selected, (eventually) providing values for all possible P_{ij}.

Table 4.3 shows the distribution of tickets for this example, where destinations can be considered the same as cores or threads, because there is only one thread per core and no idle cores. Tickets are awarded according to the past performance of thread Θ_g=300.

- Thread 300 cannot move to core 1 (its current location) or 0 (it is in the same node), so both cores get 0 tickets.

Table 4.2: Thread performance for the example of Table 4.1.

–	100	101	200	201	300	301
P	1.9	3.1	**0.9**	2.1	3.3	8.1
$\widehat{P}$	0.76	1.24	0.6	1.4	**0.58**	1.42

- Cores 2 and 3 get B_2 tickets, since there is no past information of thread 300 on node 1.

- Cores 4 and 5 get B_3 tickets, because performance of thread 300 was better on node 2 than on the current node.

Tickets are then awarded considering the past performance of the threads that are currently executing on each particular core, when executed previously on node 0, the node currently hosting thread 300.

- Core 2 gets B_6 tickets because thread 100 performed better on node 0.

- Core 3 gets B_5 tickets because thread 301 has no previous performance information on node 0.

- Core 4 gets B_4 tickets because thread 101 performed worse on node 0.

- Core 5 gets B_5 tickets because thread 201 has no previous performance information on node 0.

Thus, 21 tickets were awarded, and

- Thread 300 has 6/21 chances of migrating to core 2 and being interchanged with thread 100. This would be favourable to thread 100 and unknown to thread 300.

- Thread 300 has 6/21 chances of moving to core 5 and being interchanged with thread 201. This would be unknown to thread 201 and favourable to thread 300.

- Thread 300 has 5/21 chances of migrating to core 4 and being interchanged with thread 101. This would be detrimental to thread 101 and favourable to thread 300.

Table 4.3: Ticket distribution for the example of Table 4.2.

thread (core)	P_{i0}	P_{i1}	P_{i2}	tickets
100 (2)	2.5	**1.9**	2.9	$B_2 + B_6$ = 2+4
101 (4)	2.7	1.8	**3.1**	$B_3 + B_4$ = 4+1
200 (0)	**0.9**	1.4	–	0
201 (5)	–	1.6	**2.1**	$B_3 + B_5$= 4+2
300 (1)	**3.3**	–	6.3	0
301 (3)	–	**8.1**	5.7	$B_2 + B_5$=2+2

– Thread 300 has 4/21 chances of going to core 3 and being interchanged with thread 301. This would be detrimental to thread 301 and unknown to thread 300.

Once all tickets are awarded, Θ_g is chosen in a lottery. The interchange can be performed when Θ_m and Θ_g are chosen, migrating both threads to each other cores. Note that this is a small example, in a real situation with more threads and nodes, the probability differences among the possible migrations would be larger.

4.3 Migration tool

In order to obtain the three parameters in P_i and P_{ij}, the dynamic runtime information about the behaviour of the code, the same mechanism using PEBS described in sections 2.2 and 3.3 is used. This information includes, for load operations, the latency in which the data is served, as well as information about the memory level from where the data was actually read. Also, information about floating point operations can be captured each time a certain number of instructions has been executed. All these events are obtained in a sampled way, so each point in the 3DyRM or i3DyRM corresponds to a sampled event. The memory latency assigned to that point in the model is given by the mean latency of memory events captured along the previous time interval. So, if the memory events are captured in a rate close to that of the floating point information, each point will have a close approximation of the mean latency in such time interval.

The migration tool captures events for all the processes executed in the system. It can store performance information for each thread in the system as needed for IMA, IMAR or IMAR^2. It can pin any thread to an specific core, and perform migrations guided by the algorithms.

4.4 Case Studies: SDOT and SAXPY

In this section the IMA algorithm is tested using a couple of examples. The experiments presented in this section were carried out on our Xeon Server Y (see section 1.2 for more details). Each processor has a 20 MB shared L3 cache. The main memory is divided into two 32 GB cells, `cell0` and `cell1`. Each processor is associated to one of these cells. All executions were carried out with 16 threads, and the Hyper-Threading capability was disabled. The system runs a Linux kernel 3.10.

4.4.1 The SDOT and SAXPY routines

In our experiments, two single precision Level 1 BLAS routines, SDOT and SAXPY, were used. These routines were selected because their behaviour is well known and to fix the reproducibility of results.

- The SDOT operation computes the dot product of two real vectors in single precision:

  $\mathrm{s} \leftarrow x^\top \cdot y = \sum x(i) * y(i)$

- The SAXPY operation computes a constant *a* times a vector *x* plus a vector *y*. The result overwrites the initial values of vector *y*:

  $y \leftarrow a \cdot x + y$

Both operations work with strided vectors. Two values, named *incx* and *incy*, are used to specify the stride between two consecutive elements of vector *x* and vector *y*, respectively. Different strides are used to change the behaviour of the codes in terms of memory accesses.

4.4.2 The implementations

To place segments of each vector in different memory cells, the `libnuma` library [32] has been used. Each vector has been divided into 16 segments, one for each execution thread, so each one can be allocated to a specific memory cell using `numa_alloc_onnode()`. Furthermore, each thread can be assigned to a specific core using `sched_setaffinity()`. In this way, different configurations have been tested:

- IDEAL: Each thread operates with the vector segments it needs in its local memory.

- CROSSED: Each thread operates with the vector segments it needs in its remote memory.

- ALL_IN_0: All the segments are placed in `cell0`.

- ALL_IN_1: All the segments are placed in `cell1`.

4.4.3 Selection of parameters

In order to show the use of our strategies, in these case studies, the following values have been selected for the parameters that rule the algorithms: $T = 1$ s, as the time between migrations, and $\Theta = 2$, the number of thread to be migrated. The value of T was selected as a trade-off between efficiency and overhead. We found that, in our system, this time is good to detect the effects of the last migration from the monitoring information. Also, the migration itself does not interrupt the execution of the codes. The value of Θ was selected to make transitions between consecutive migrations as smooth as possible. In addition, four configurations of the weighting factors of equation 4.1 were considered to study the relative importance of the three components of P_i. Using the notation of the IMA, these configurations are IMA$[1;2;1,1,1]$, IMA$[1;2;2,1,1]$, IMA$[1;2;1,2,1]$ and IMA$[1;2;1,1,2]$. Since T and Θ remain the same they can be called 111, 211, 121, and 112 according to the values of α, β and γ, respectively. These configurations cover both the case in which the importance of the parameters is balanced, and the one in which one of the parameters is more relevant than the others. In addition, these values guarantee a fast computation of P_i. Each set of experiments was performed 5 times and average execution time was extracted. The system used was our Xeon Server Y (see Section 1.2.2 for details).

4.4.4 Results for IMA

Results obtained for the SAXPY kernel with 16 threads using stride 4 are shown in Figure 4.1. In order to be fair, both of IMA, W_RBEST and W_RWORST, described in Section 4.2.1, are compared with other two strategies: FIXED, that means that all threads remain executing always in the same core (no migrations at all), and FREED that means that the OS migrates threads following its own mechanisms. Horizontal axis indicates the weighting of the factors in Equation 4.1. The size of x and y were fixed in all the experiments to $5 \cdot 10^7$ elements, that is, $2 \cdot 10^8$ bytes, far larger than the size of L3 cache. Each experiment was repeated 700 times.

As the stride increases, less operations are performed. For stride 8, for example, only half of the vector takes part in the operation compared to stride 4. Nevertheless, from stride 4 to stride 16 (see Figures 4.1, 4.2, and 4.3 for SAXPY, and Figures 4.4, 4.5, and 4.6 for SDOT) execution times remain almost the same. This is due to the management costs of the memory hierarchy. In the Sandy Bridge architecture, the cache line size is 64 bytes, which means it can hold 16 floats. Furthermore, the processor always reads two consecutive cache lines from main memory, so it brings 32 floats from the cache. This means that from stride 4 to stride 16 the system will transfer from main memory to the cache the same amount of data, essentially the whole vectors x and y. So, with stride 16 only one float is needed per cache line for each vector, but the system will still move the full cache lines, 128 bytes. As a consequence, these codes are memory bound, and therefore, their execution time is limited by memory accesses, and it is not reduced, even though they are executing less floating point operations.

In the IDEAL configuration each thread is using the memory module closest to itself, which should be the best case for memory access and should present lower latencies. In the CROSSED configuration each thread is using the memory opposite to itself. As expected, the ALL_IN_0 and ALL_IN_1 configurations show the worst results. This is because all threads access the same memory cell, which produces bus conflicts and saturation. In the CROSSED configuration data need more time to reach its destination, but the number of read conflicts is similar to the IDEAL configuration. In the ALL_IN configurations, the cell where the data

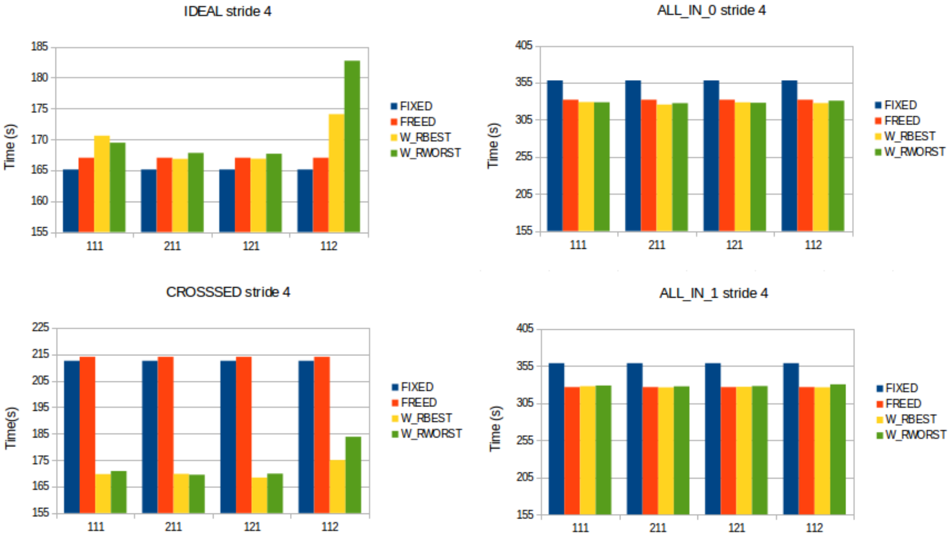


Figure 4.1: Execution times of SAXPY with stride 4.

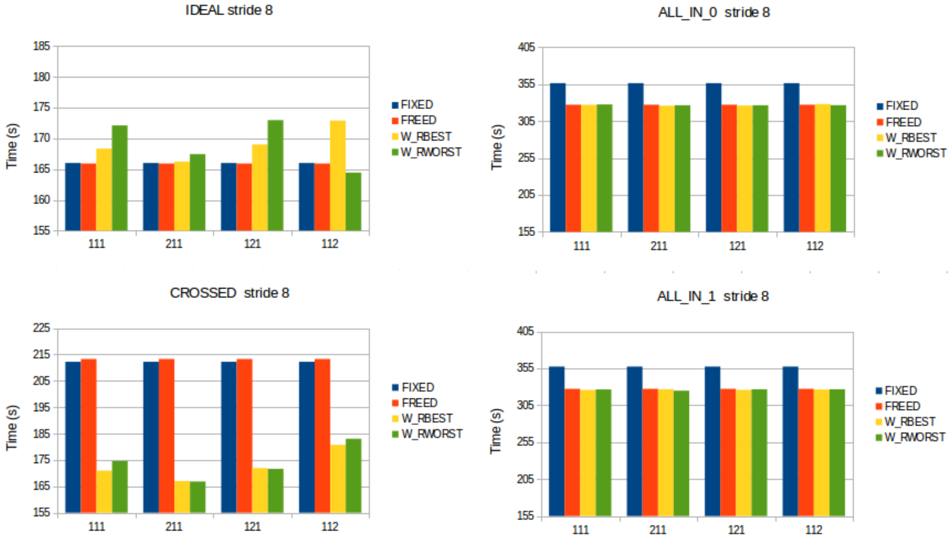


Figure 4.2: Execution times of SAXPY with stride 8.

.

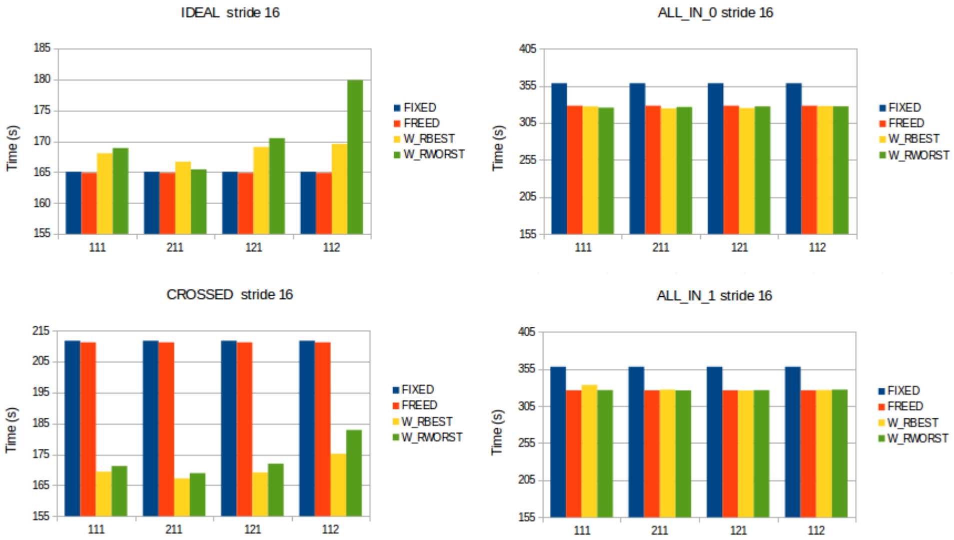


Figure 4.3: Execution times of SAXPY with stride 16.

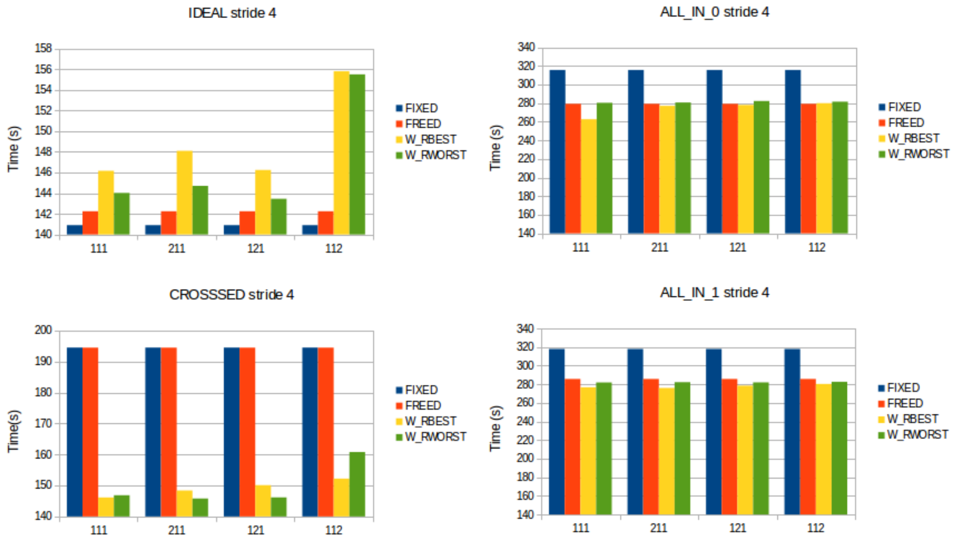


Figure 4.4: Execution times of SDOT with stride 4.

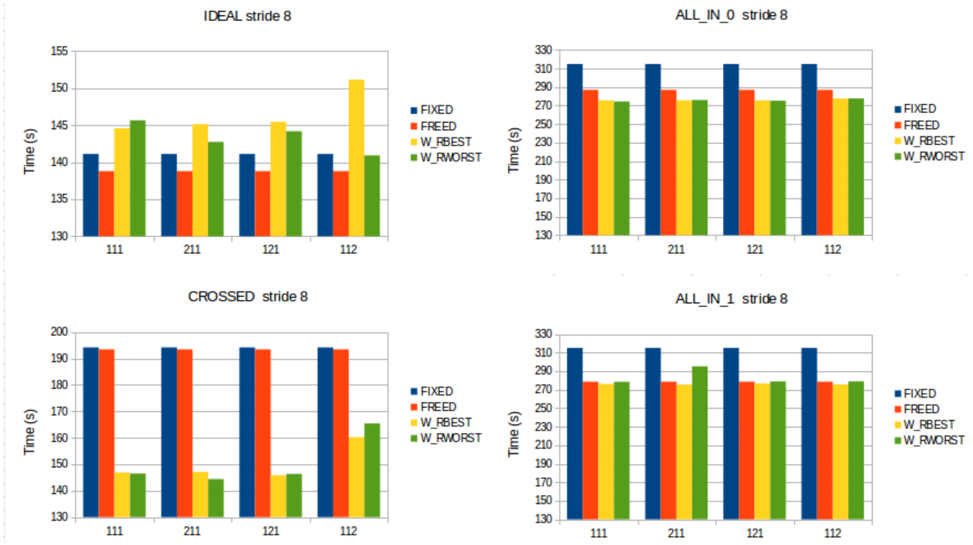


Figure 4.5: Execution times of SDOT with stride 8.

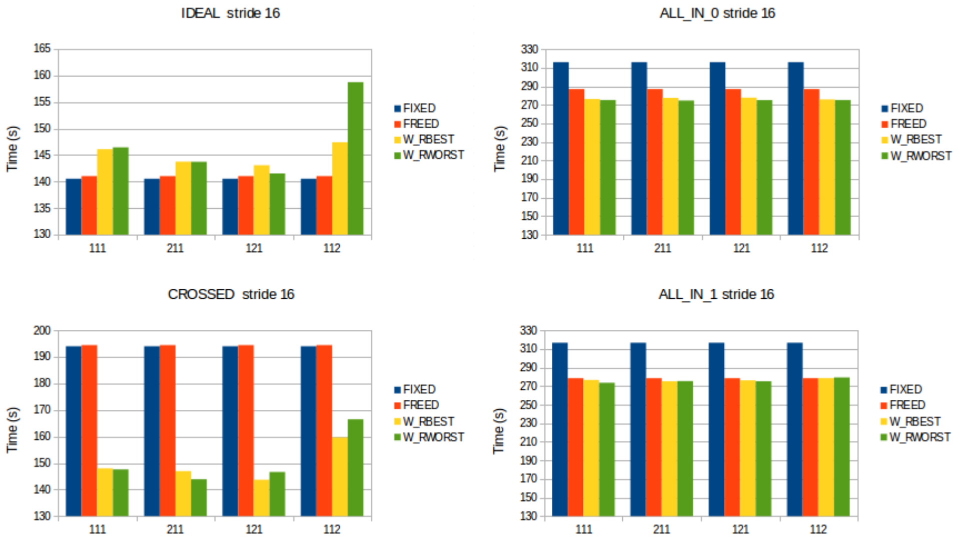


Figure 4.6: Execution times of SDOT with stride 16.

is stored shows better behaviour than its opposite, but the overall performance is diminished. In fact, for ALL_IN configurations, threads in the same cell as the data finish their execution before threads in the opposite cell. This states the importance of balancing the memory use.

In the IDEAL case there is no gain from using our migration mechanisms. This is because, the IMA algorithm always migrate threads every T milliseconds, and they do not stop even when a good configuration is reached. Since the IDEAL case already starts with the best possible configuration, any thread movement is basically detrimental.

In the ALL_IN cases the threads placed in the processor with affinity to the data finish their computation before the threads in the other processor. This is why migrations help performance, since they balance the load between the processors and make them finish roughly at the same time. When only the OS is in charge of migrations (FREED case), one of the processors finishes first, but the remaining threads are then balanced between the two processors, balancing the load in the end.

The CROSSED case is the one in which more gain is obtained by our migration strategies. During the execution, all threads are balanced and at the same distance from data, so they take roughly the same time to finish. This means that load is balanced and therefore the OS by itself does not perform migrations at all. Nevertheless, it takes longer to finish computation compared to the IDEAL case due to the high memory latency. Using our migration strategies, threads are placed closer to their data and execution time gets closer to the IDEAL case. In the CROSSED configuration is where choosing the correct strategy for thread migration is more important.

The migration configurations of the scaling factors (111, 211, 121, and 112) affect the results of both migration algorithms. Configuration 112 gets the worst results. This is because it amplifies the importance of the OI, and this factor should be roughly the same for all threads, since they perform the same number of operations. As a consequence, the algorithms are essentially migrating threads randomly. Configuration 121 gives more importance to GFLOPS. Nevertheless, there is a problem with the floating point operations (FP_OPs) hardware monitoring in the Intel architectures, as stated in section 3.4.2. This makes results for configuration 121 slightly worse than those for 211. Configuration 211 gives more importance to latency. It achieves the best results with these codes, since memory latency is the determining factor of their performance. W_RWORST performs better than W_RBEST with the SDOT kernel, while with the SAXPY kernel results are similar for both strategies. The W_RWORST should reach the IDEAL configuration from the CROSSED one faster, since it starts moving the worst performing threads first and, once threads are placed in the proper cell, they perform better and tend to not be migrated. Anyway, since neither algorithm stops migrating once a good configuration is reached, both perform similarly.

4.5 NAS Case Studies

NPB-OMP benchmarks were used to study the effect of the memory allocation. These benchmarks are well suited for multicore processors, although they do not greatly stress the memory of large servers. In this section a series of tests based on these benchmarks are presented. These tests are designed to stress NUMA memories and allow to carry out experiments with algorithms IMAR and IMAR^2. To simulate the effects of NUMA memory allocation, different memory stress situations were forced using the numactl tool [32], which allows to define the memory cell to store data and to pin the threads to specific cores or processors.

4.5.1 NAS implementations

We designed an experiment where four instances of the NPB-OMP benchmarks are executed concurrently in a multiprocessor system, and the placement of each one can be directly controlled. Each benchmark instance was executed in one multi-threaded process. The system used was our Xeon Server Z (see Section 1.2.2 for details). Each benchmark was executed with just enough threads to fill one node. Thus, each process could have its execution threads pinned to any node and its data assigned to a selected memory cell. Different memory placement scenarios can be established by executing as many process as nodes. We tested the following options:

– FREE test: The benchmarks started execution at the same time, and the OS controlled memory and thread placement.

– DIRECT test: Each benchmark had its threads fixed to one node and preferred memory set to the same cell.

- CROSSED test: Each benchmark had its threads fixed to one processor and preferred memory set to a different cell. When more than two cells were considered, there were several possible combinations. The configuration used in the case study with four cells was:
    - threads in node 0 had their data in cell 1,
    - threads in node 1 had their data in cell 0,
    - threads in node 2 had their data in cell 3, and
    - threads in node 3 had their data in cell 2.
- INTERLEAVE test: Each benchmark had its threads fixed to one node and memory set to interleave, with each consecutive memory page set to a different memory cell in a round robin fashion.

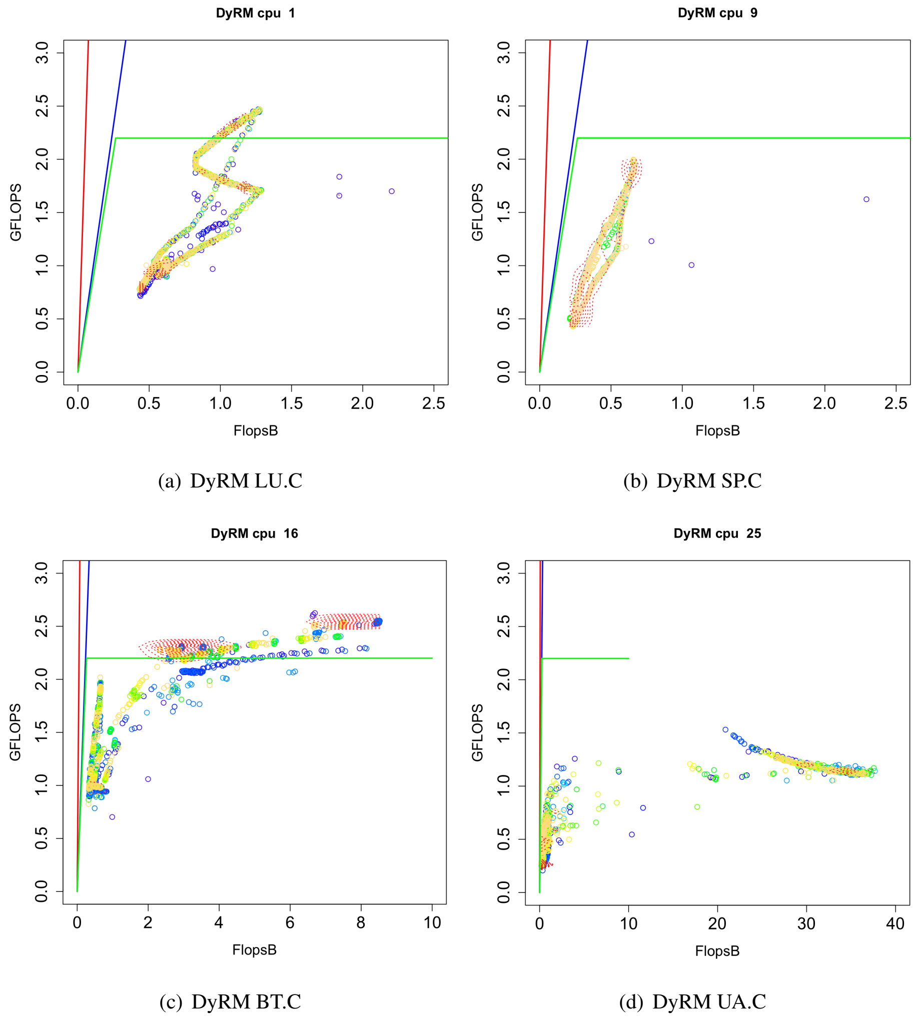
Four class C NPB-OMP codes were selected to be considered: `lu.C`, `sp.C`, `bt.C` and `ua.C`. This selection was made according to two main criteria: To consider codes with different memory access patterns and with different computing requirements.

- The DyRM model (see Figure 4.7) was used to select two benchmarks with low `flopsB` (`lu.C` and `sp.C`) and two with high `flopsB` (`bt.C` and `ua.C`).
- Since the execution times of these codes are similar, they remain in concurrent execution most of the time. This helps studying the effect of thread migrations.

All benchmarks were compiled with gcc and O2 optimisation.

4.5.2 Baseline results

In this section, the results of the execution of our tests are shown. The effects of the memory placements in the execution of the NPB-OMP benchmarks are evaluated, without migration strategies, and these results are used as a baseline to evaluate IMAR and IMAR^2. These tests were executed on system Z using only nodes 0 and 1, to see the behaviour on 2 nodes,



(a) DyRM LU.C

(b) DyRM SP.C

(c) DyRM BT.C

(d) DyRM UA.C

Figure 4.7: DyRM for selected NAS Benchmarks. Note that `lu.C` and `sp.C` have much lower FlopB than `bt.C` and `ua.C`.

Table 4.4: Baseline times for dual NAS on System Z with 2 nodes.

	Time (s)			
test	free	direct	interleave	crossed
lu.C	218,05	210.21	533.95	1201.93
sp.C	265.91	268.246	697.75	1680.43
bt.C	195.72	181.24	279.46	443.6
ua.C	203.84	190.6	368.39	659.71
sp.C	205.77	264.9	911.44	1683.93
ua.C	224.56	191.02	376.03	677.99
sp.C	204.84	265.48	743.09	1695.49
bt.C	206.84	182.99	284.35	463.11
lu.C	221.15	202.99	545.75	1220.02
bt.C	198.32	188.35	282.99	455.83
lu.C	212.61	209.88	541.61	1203.03
ua.C	196.63	190.86	368.26	664.38

and using the 4 nodes. The NAS benchmarks were combined when executed in 2 nodes as two different instances (6 combinations were considered, **lu.C/sp.C**, **bt.C/ua.C**, **sp.C/ua.C**, **sp.C/bt.C**, **lu.C/bt.C**, and **lu.C/ua.C**). The execution times obtained for these benchmarks are shown in table 4.4. These results show that, for the FREE case, when threads and memory are left to be freely placed by the OS (limited to 2 nodes, not using the 4 available ones), execution times are, in most cases, worse than the DIRECT case. Since the NAS benchmarks perform reasonably well on multicores and they do not stress the memory, on the FREE test the OS placement of threads and memory performs well, although it is not as good as the DIRECT case.

The exception is with benchmark `sp.C`, specially when paired with `bt.C` or `ua.C`. This benchmark is memory intensive, and the OS seems to accelerate its execution by making the opposite application slower. For the rest of the cases, placing the memory and the execution threads in the same node seems to be the best option. Interleaving the memory does not get good results, but by far the worst case is the CROSSED test, where memory and threads are in opposite nodes.

When 4 nodes are considered, each test was executed on the four nodes, combined as four processes of the same code that produced four combinations (**4 lu.C**, **4 sp.C**, **4 bt.C**, and **4 ua.C**), and four processes of different codes, that produced one combination (**lu.C/sp.C/bt.C/ua.C**). Every test was executed five times and the mean execution times are shown in Table 4.5. The times for all benchmarks of **lu.C/sp.C/bt.C/ua.C** are shown, whereas, for considerations of space, only the times of the fastest and slowest instances are shown for the four equal benchmarks.

Results are similar to the ones of the 2 nodes case. The FREE test, where the OS placed threads and memory freely, performs reasonably well, although inferior to the DIRECT case. `sp.C` is inferior to the DIRECT test, but only when executed with other codes in the **lu.C/sp.C/bt.C/ua.C** combination. For that case, when benchmarks `ua.C` and `bt.C` finish execution in the FREE test, the OS is free to place `sp.C` threads in other processors to balance the load, which leads to a faster execution. For the other cases, placing memory and execution threads in the same node appears to be the best option, while interleaving memory does not produce good results. As expected, by far the most inferior case is the CROSSED test, where memory and threads are on different nodes.

Thus, the DIRECT case is the best option, although it has some load balancing issues that can decrease the global performance, and the CROSSED test is the worst case, as expected.

4.5.3 Study of traces

Before presenting the execution times of the benchmarks when the IMAR and IMAR^2 algorithms are applied, a few examples of the behaviour on specific cases are warranted. The migration tool can be configured to dump the PEBS trace to a file, which can be read by a performance visualization tool, such as in [46]. Thus, the evolution of the performance of

Table 4.5: Baseline times for 4 NAS on System Z with 4 nodes.

	Time (s)			
test	free	direct	interleave	crossed
lu.C	220.24	210.00	428.41	1221.05
sp.C	235.53	267.89	557.39	1698.36
bt.C	201.69	180.77	260.46	500.037
ua.C	197.03	190.26	316.26	759.17
fastest lu.C	213.09	209.99	444.09	1265.46
slowest lu.C	215.84	212.20	452.15	1278.86
fastest sp.C	267.80	265.29	511.15	1848.41
slowest sp.C	287.49	267.71	763.88	1864
fastest bt.C	181.27	180.74	242.52	452.47
slowest bt.C	185.37	182.29	246.90	453.13
fastest ua.C	194.51	189.36	303.76	677.31
slowest ua.C	203.54	190.46	313.59	684.70

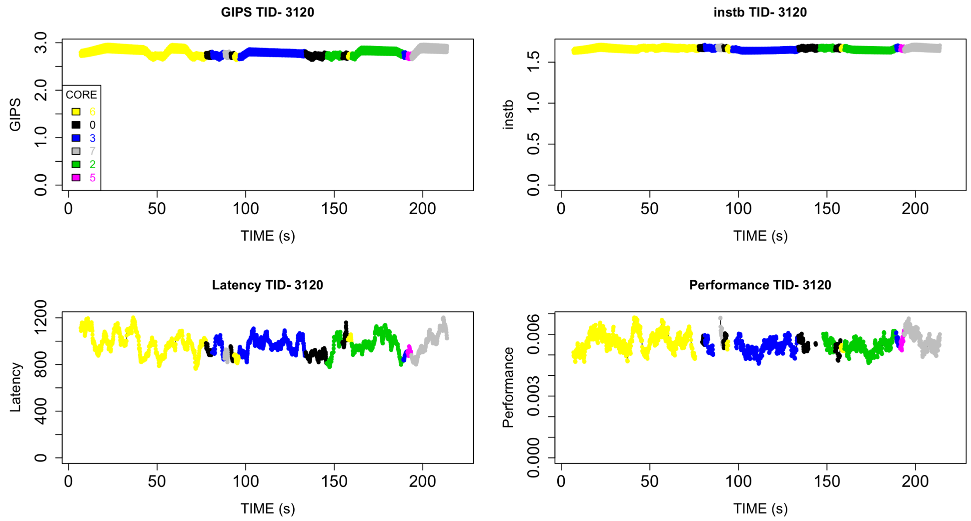


Figure 4.8: Evolution of performance for one thread of the **4 lu.C** configuration for the DIRECT case. The thread runs in node 0.

each thread, in terms of P_{ij} and its components (equation 4.4), through the execution of the benchmarks can be plotted. In these cases the parameters α, β, and γ are set to 1 to make the figures easier to interpret. Figures 4.8 and 4.9 show the performance of a thread of the **4 lu.C** benchmark in the DIRECT and the CROSSED configurations, respectively.

In these figures, different line colors represent different cores, and a change in color represents a migration of the thread. To better visualise the changes, we used a frame average of 50 measurements, corresponding to measurement every 1.5 seconds. This frame average implies that performance changes between migrations are not instantly visible, but usually take the form of peaks and valleys. In Figures 4.8 and 4.9, migrations were performed by the OS among cores in the same node, so performance does not vary greatly during execution. As expected, performance is lower on the CROSSED test, with more migrations involved.

In Figures 4.10 and 4.11 the performance of two threads during the execution of the **4 lu.C** combination in the CROSSED test and IMAR migrations are shown. For example, in Figure 4.11, it can be seen that performance increases and approaches the DIRECT case due to the IMAR migrations. Using IMAR, migrations take place between nodes, so they influence the performance more than in the case of Figure 4.9. Peaks in the graph of the same color,

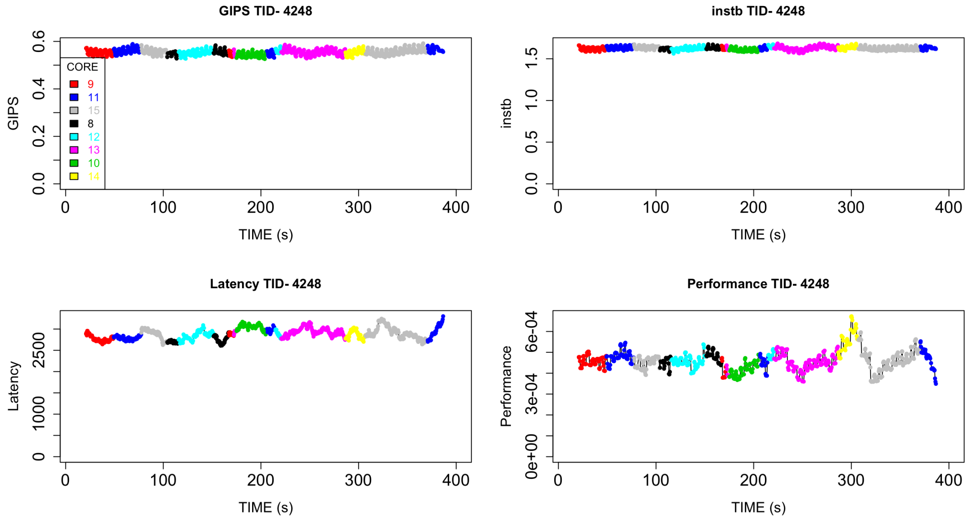


Figure 4.9: Evolution of performance for one thread of the **4 lu.C** configuration for the CROSSED case. The thread runs in node 1.

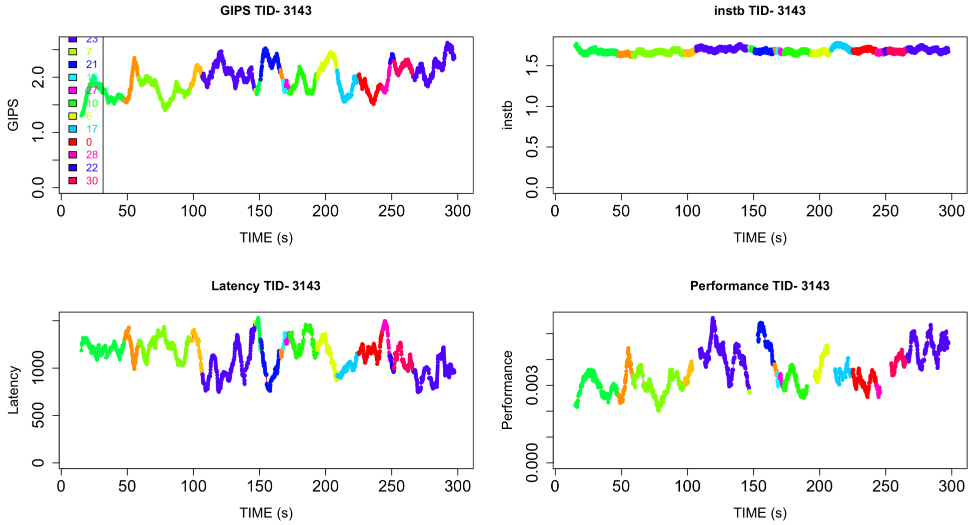


Figure 4.10: Evolution of the performance for one thread of the **4 lu.C** configuration for the CROSSED test with IMAR migrations, thread 3143.

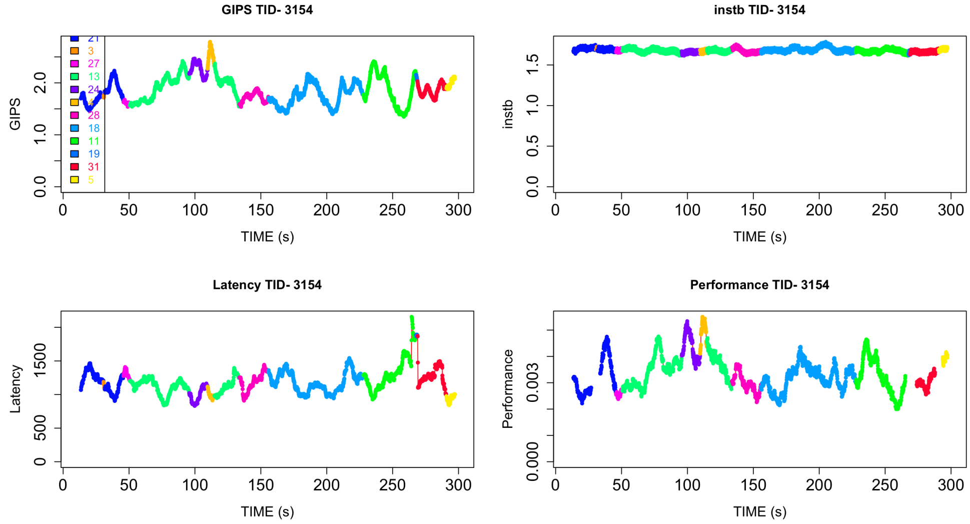


Figure 4.11: Evolution of the performance for one thread of the **4 lu.C** configuration for the CROSSED test with IMAR migrations, thread 3154.

are likely due to migrations of other threads that influence the single-color thread, whereas peaks with a color change are due to migrations of the thread itself. Note that migrations usually occur after a performance dip, because the thread was chosen to be among the worst performing by the IMAR algorithm. For example, in Figure 4.11, the migration after 250 seconds is apparently due to an increase in memory latency.

The performance of two threads during the execution of the **4 lu.C** combination in the CROSSED test and IMAR^2 migrations ($\omega = 0.97$) are shown in Figures 4.12 and 4.13. The tendency towards increasing performance is clear, because the rollbacks reduce the number of migrations. There are less pronounced variations in performance than in the IMAR case, due to the varying T and rollback strategies. In Figure 4.12, a dip in performance of thread 109565 close to 150 seconds triggers a migration from core 3 back to core 25, that is, a rollback. In Figure 4.13, a migration from core 13 (in node 1) places the thread in core 6 (in node 0), and subsequently there are rollbacks around 70, 130, and 260 seconds, which indicate that thread 109553 was placed in an efficient node, and it is inefficient to move it. The IMAR^2 algorithm explores all possible placements for all threads, and so counter-productive migrations can be performed, but including rollback allows their effects to be minimised.

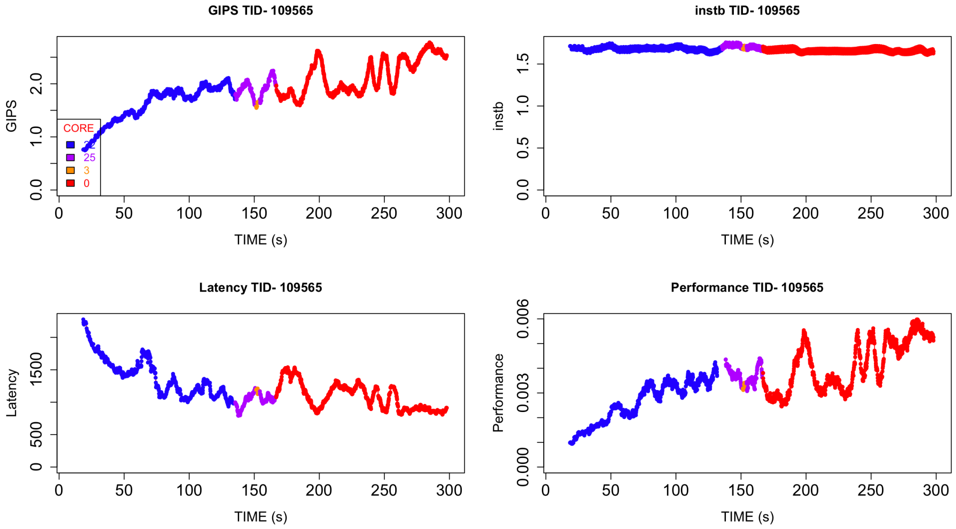


Figure 4.12: Evolution of performance for one thread of the **4 lu.C** configuration for the CROSSED test with IMAR^2 migrations, thread 109565.

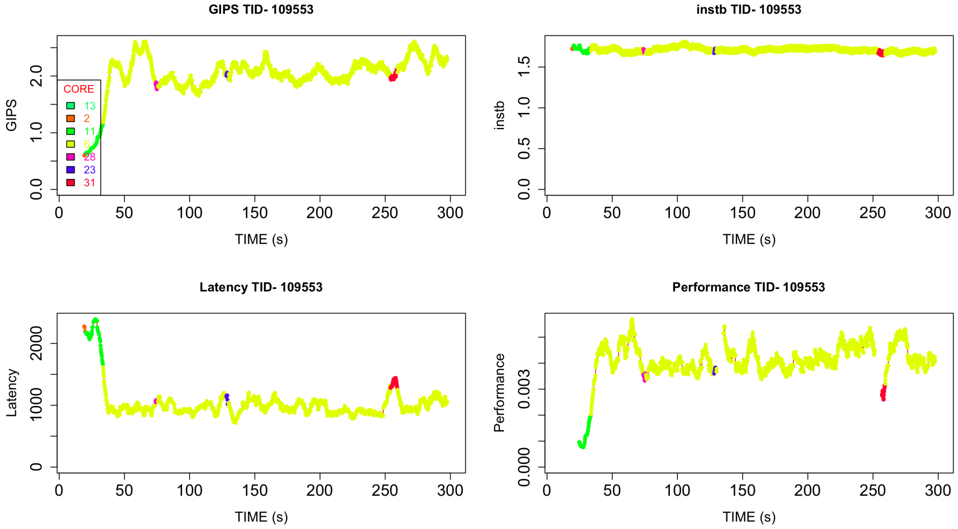


Figure 4.13: Evolution of performance for one thread of the **4 lu.C** configuration for the CROSSED test with IMAR^2 migrations, thread 109553.

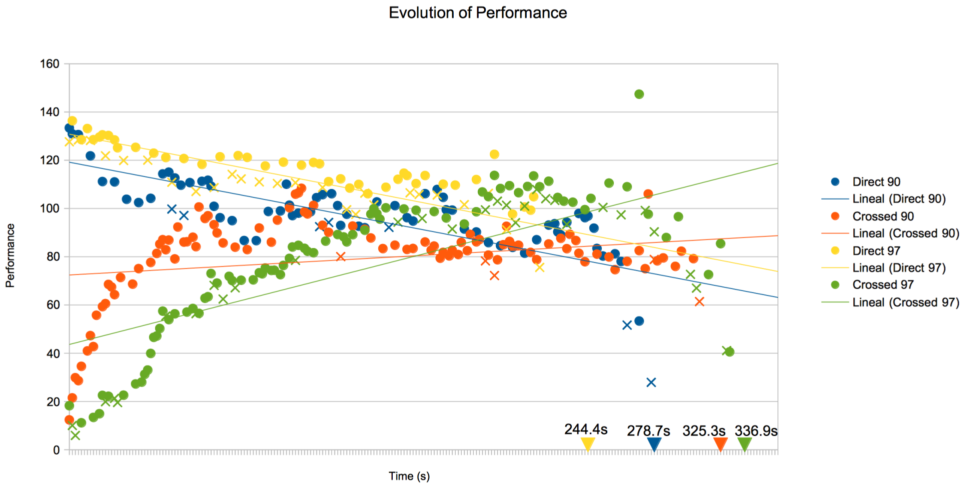


Figure 4.14: Evolution of performance for the **4 lu.C** configuration for the CROSSED and DIRECT cases with IMAR^2 migrations. A linear approach for each case is also shown.

The algorithm tries other node placements for the thread, computing the whole performance record (moving to core 28, node 4, to core 23, node 3, etc.) and checking for behaviour changes, but always returns the thread to core 6 in node 0.

An example of migration timing in the **4 lu.C** combination for the CROSSED and DIRECT tests with IMAR^2 migrations is shown in Figure 4.14, along with a lineal approximation to show the tendency of the performance evolution. Thresholds $\omega = 0.90$ and $\omega = 0.97$ were considered, and the performance record for the whole system is shown, where a circle represents a migration, a cross represents a rollback, and triangles mark the execution time of each test. This graph is from a single execution of each case for each value of ω.

In DIRECT cases, performance remains higher with $\omega = 0.97$ through the executions, due to rollbacks, since migrations are counter-productive in this case. When performance dips, a rollback is executed (a yellow cross in the figure) and it recovers.

In the CROSSED configurations, where migrations are initially productive, when all threads are in inefficient placements, performance increases faster with $\omega = 0.90$ than with $\omega = 0.97$. With $\omega = 0.90$, no rollbacks are performed during the first minute, while rollbacks with $\omega = 0.97$ are counter-productive, since they make the process slower when approaching the best placements. Nevertheless, once performance is high enough, and more threads are correctly placed, the $\omega = 0.97$ case helps keep performance high with rollbacks, whereas when $\omega = 0.90$, migrations continue even once a good configuration is obtained.

4.5.4 Case study on two nodes with IMAR

All figures in this next sections show the experimental execution times performing migrations, by IMAR, as a proportion of the baselines times of each test (FREE, DIRECT, INTERLEAVE and CROSSED), expressed as a percentage. A percentage greater that 100 means a worse execution time, while a result under 100 shows a better execution time. A special case is shown for the OS, where the DIRECT, INTERLEAVE and CROSSED tests are modified to fix only the memory placement, letting the OS select thread placement. These tests were made to see if the OS was able to detect where each thread had its data and place the thread accordingly.

The mean results of using the IMAR algorithm with 2 nodes and different values of T and α, β and γ are shown in Figure 4.15. First, it must be noted that the OS mainly migrates threads for load balance; so there are few migrations done until one of the benchmarks ends. Also, for all the modified tests for the OS, it does not seem to take into account the memory placement to pin or migrate threads, in fact, the initial thread placement seems to be done at random. This means that, for the DIRECT case, the performance is being decided by the initial thread placement, which is unlikely to be the ideal one, therefore, it implies worse results, and with a greater variability, than those of the guided migrations. In the same way, results are better for the CROSSED test, because the initial placement is also unlikely to be the worst one. Actually, results for the DIRECT and CROSSED tests with the OS are similar in absolute execution times values, since conceptually, once there is no thread pinning, they are almost the same test (both pin the data of each process to one of the nodes). In the INTERLEAVE test, where the initial placement matters less, the OS performs better. In essence, the OS works well when it can decide both the memory and threads placements, the FREE test, but it is not able to detect a forced memory placement and act accordingly.

For the IMAR algorithm, in general, performing any migrations on the FREE test or on the DIRECT test results in a worse performance compared to the baseline. This is mainly because in both cases the starting configuration is close to the ideal one. Loss of performance

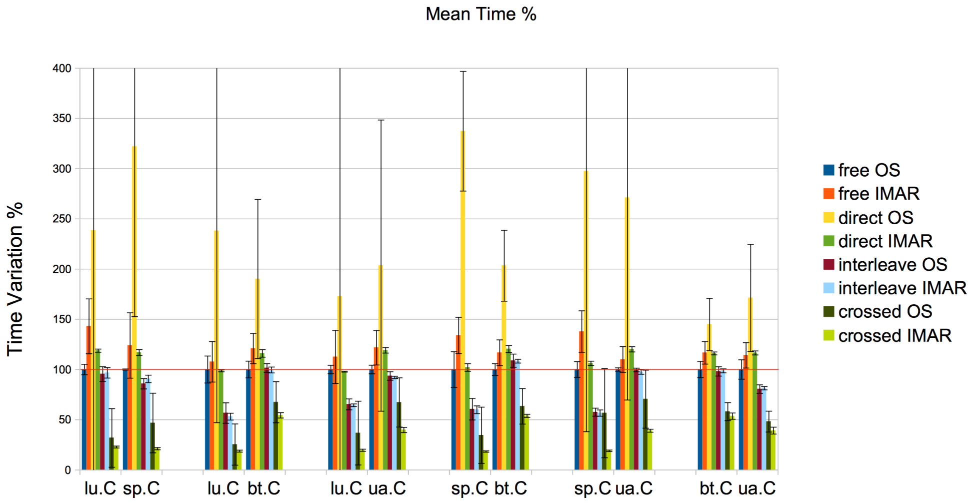


Figure 4.15: Mean results for all test with 2 nodes.

is greater on the FREE case where the Operating System may not place all the data of a benchmark in one node, which would make our algorithm less accurate on its migrations.

On the DIRECT case, our algorithm seems to favour memory intensive applications, like **lu.C** and **sp.C**. When these applications are paired with others with high `instB`, like **bt.C** and **ua.C**, they suffer less performance loss, and may even gain performance (like **lu.C** on **lu.C/bt.C** and **lu.C/ua.C**). For instance, results for **sp.C/bt.C** show better performance for **sp.C** (Figure 4.16) than for **bt.C** (Figure 4.17). In these figures, for **sp.C** in the DIRECT case, there is little loss of performance but at the cost of decreasing the performance of **bt.C**.

On the INTERLEAVE case, migrations give better performance. Results with the OS, which hardly performs any migrations during execution, are close to those with our algorithm, which indicates that when memory is interleaved the thread affinity is not considered important.

On the CROSSED case, migrations lead to better performance. This is to be expected since the initial configuration was the worst possible, so any migration should lead to a better configuration. Performing more migrations (every 4 seconds instead of every 8 seconds) give better results, contrary to the FREE or DIRECT cases.

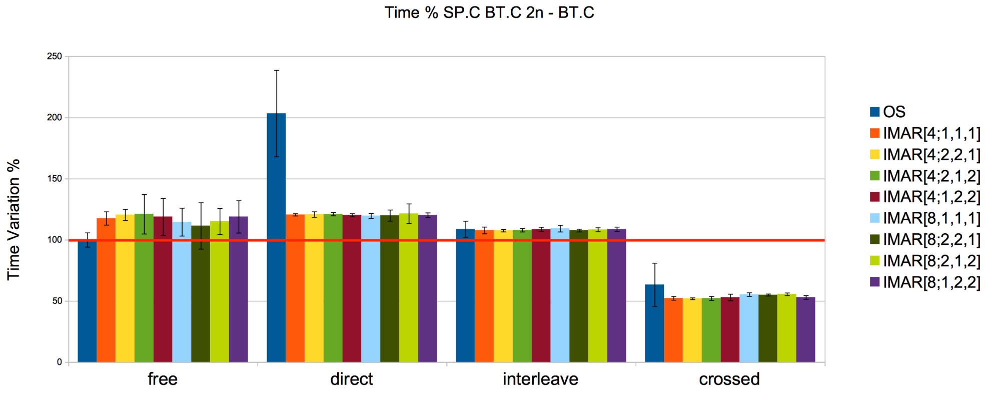


Figure 4.16: Variations of sp.C for **sp.C/bt.C** with respect to the baseline (2 nodes).

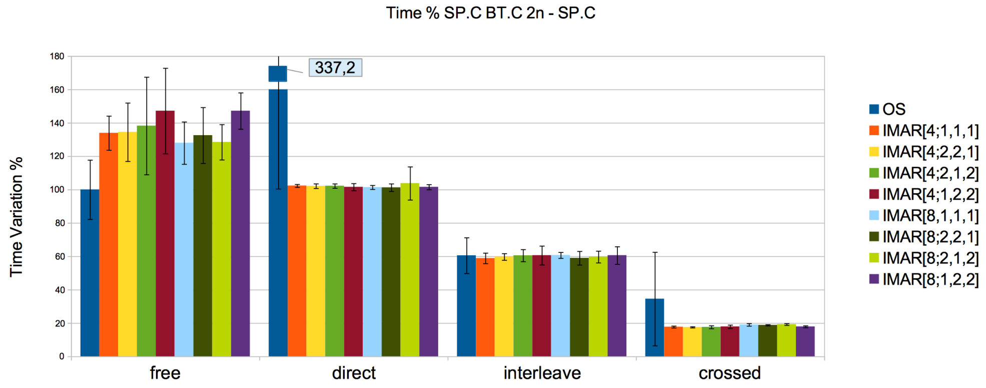


Figure 4.17: Variations of bt.C for **sp.C/bt.C** with respect to the baseline (2 nodes).

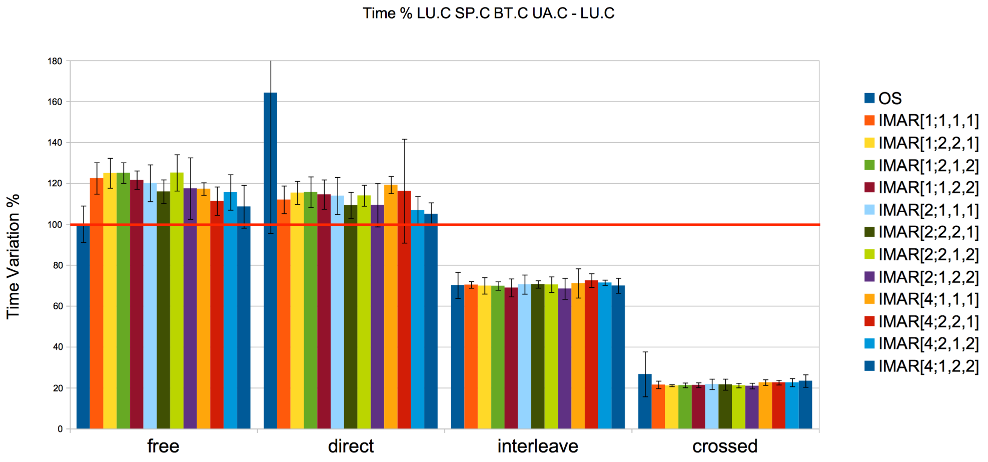


Figure 4.18: Variations of lu.C for **lu.C/sp.C/bt.C/ua.C** with respect to the baseline.

4.5.5 Case study on four nodes with IMAR

Results for 4 nodes are broadly similar to those with 2 nodes. Here we discuss variations in execution time of our tests compared to the baseline results of Table 4.5. Figures 4.18 to 4.21 show the results of one benchmark for all the tests with the **lu.C/sp.C/bt.C/ua.C** combination. Executions using IMAR with different values of T, α, β, and γ are also shown.

In this case, the effect of T, which determines the number of migrations, is critical. On most of these tests, the benchmarks use the same code, which makes comparing their performance fairer and easier. For the **lu.C/sp.C/bt.C/ua.C** combination, there is an apparent bias towards applications with low `instB` (Figs. 4.18, 4.19, 4.20 and 4.21), with superior results for **lu.C** and **sp.C** than for **bt.C** and **ua.C**. Note that **bt.C** and **ua.C** execute faster, and so must always share the system among four benchmarks, whereas **lu.C** and **sp.C** have more free cores at the end of their execution. This situation produces superior performance, in part due to frequency scaling capabilities on Xeon systems, when core frequency increases if not all cores are active.

Changing the scaling factors α, β, and γ has a slight impact on the effect of the migrations. For example, in the **lu.C/sp.C/bt.C/ua.C** combination, for **lu.C**, Fig. 4.18, configurations which give greater importance to memory latency, IMAR[$T;2,2,1$] and IMAR[$T;2,1,2$], performance is superior in the DIRECT and CROSSED tests, where data locality is more im-

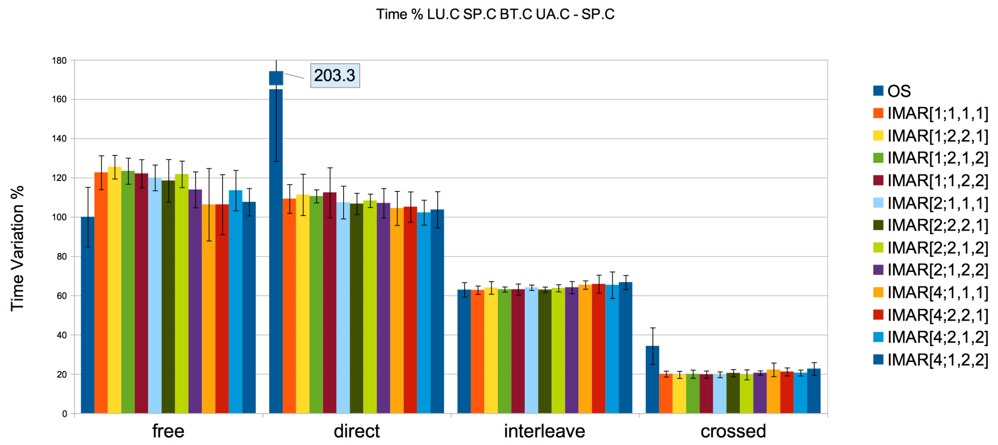


Figure 4.19: Variations of sp.C for**lu.C/sp.C/bt.C/ua.C** with respect to the baseline.

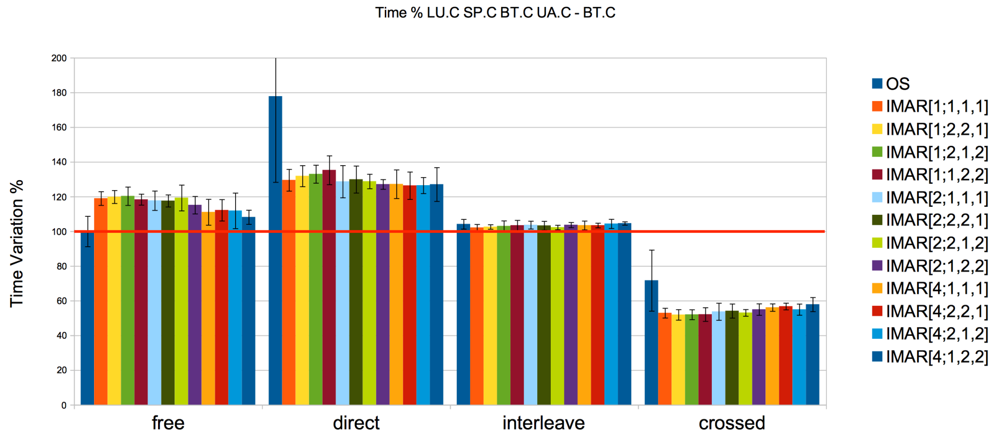


Figure 4.20: Variations of bt.C for**lu.C/sp.C/bt.C/ua.C** with respect to the baseline.

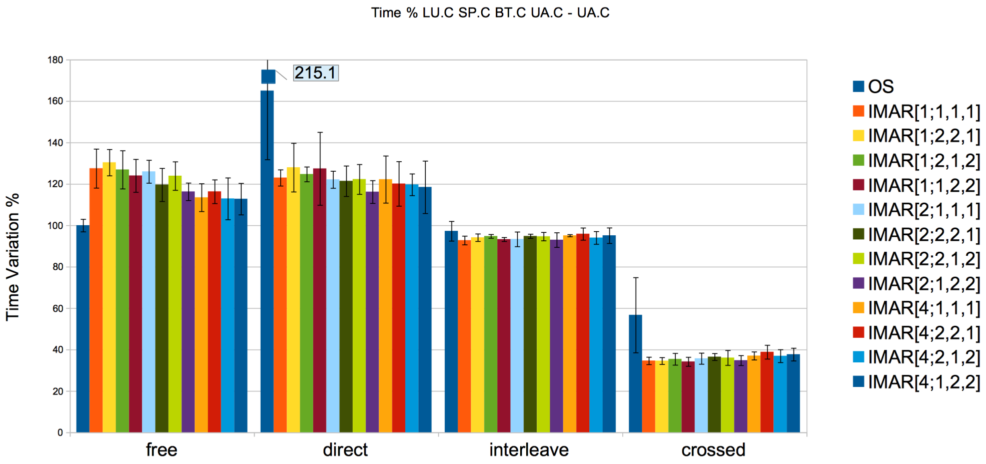


Figure 4.21: Variations of ua.C for **lu.C/sp.C/bt.C/ua.C** with respect to the baseline.

portant, and inferior in the INTERLEAVE test, where memory latency is more balanced in all nodes. Figure 4.19, corresponding to **sp.C**, shows similar outcomes to **lu.C**, since they are both memory intensive benchmarks, but with more clear influence of the migrations, since memory latency is more important. Figures 4.20 and 4.21 show less difference among configurations because latency is not so important in these cases. Nevertheless, given that all configurations take into account all the 3DyRM parameters, the differences are small, meaning the selection of the scaling factors is not critical, only needed for an extreme fine-tuning.

Figures 4.22 and 4.23 show the results for the **4 lu.C** combination. In this combination all the instances are of the same code, a **lu.C**, so results for the fastest and slowest instances are shown. While **lu.C** is a memory intensive benchmark, the best results are for configurations that prioritise `GIPS` and `instB`, not `latency`, probably because here these prioritise cache reuse and fewer main memory accesses. Since all processes compute the same operations, their `instB` should be approximately the same. This means that variations in the OI may be due to variations in the number of bytes accessed from memory (occurrences of the HC event OFFCORE_REQUEST: ALL_DATA _READ, see Section 3.3), that is, a better OI would mean better use of the caches. What is very clear in these figures is the effect of T, the lower T is, the more migrations are performed. Migrations are counter-productive in the DIRECT case, and this is shown in the figures, where the best results are with $T = 4$, but productive in

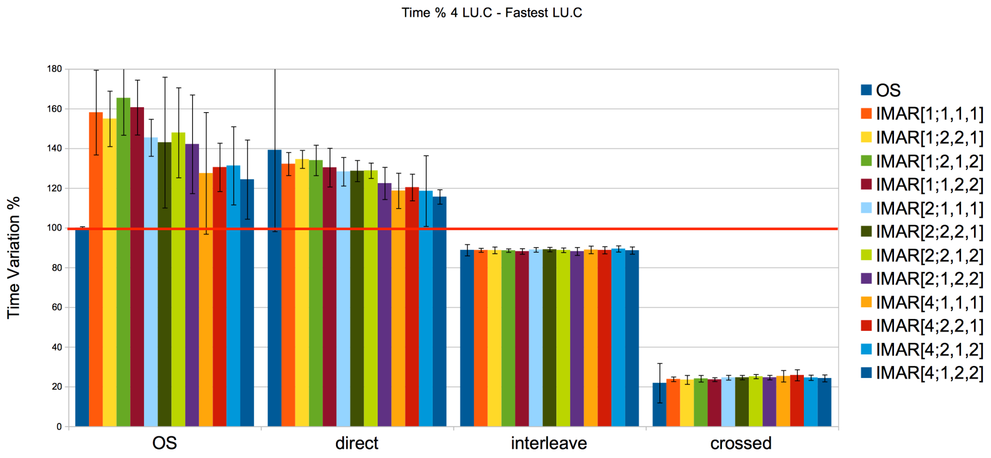


Figure 4.22: Variations of the fastest lu.C for **4 lu.C** with respect to the baseline.

the CROSSED case, although the effect is not as pronounced. To ameliorate these results the IMAR^2 can be used, as described in the next subsection.

4.5.6 Results with IMAR^2

To compare IMAR^2 with IMAR, the minimum and maximum times for IMAR^2 were set to $T_{min} = 1$ and $T_{max} = 4$, so migrations would take place at approximately the same times as in the IMAR study of the previous subsection. In general, IMAR^2 is superior to IMAR. For example, for combination **4 lu.C** (Figs. 4.24 and 4.25), as ω increases from 0.90 to 0.97, the loss of performance in FREE and DIRECT tests is reduced, while in the INTERLEAVE and CROSSED cases the behaviour of IMAR^2 remains similar to the one of the IMAR algorithm. Figures 4.26–4.29 show a closer look at the tests with only OS migration, compared to IMAR[1;1,1,1], and IMAR^2[1,4,2;1,1;0.97]. These are similar to previous figures, but the data were collated in a different way. Results are shown for each benchmark instance, for every combination, for one given test. With $\omega = 0.97$, most cases show less than a 10% loss of performance from the baseline FREE (Fig. 4.26) and DIRECT (Fig. 4.27) tests, and the performance increase from baseline INTERLEAVE (Fig. 4.28) and CROSSED (Fig. 4.29) tests are similar or superior to the IMAR case.

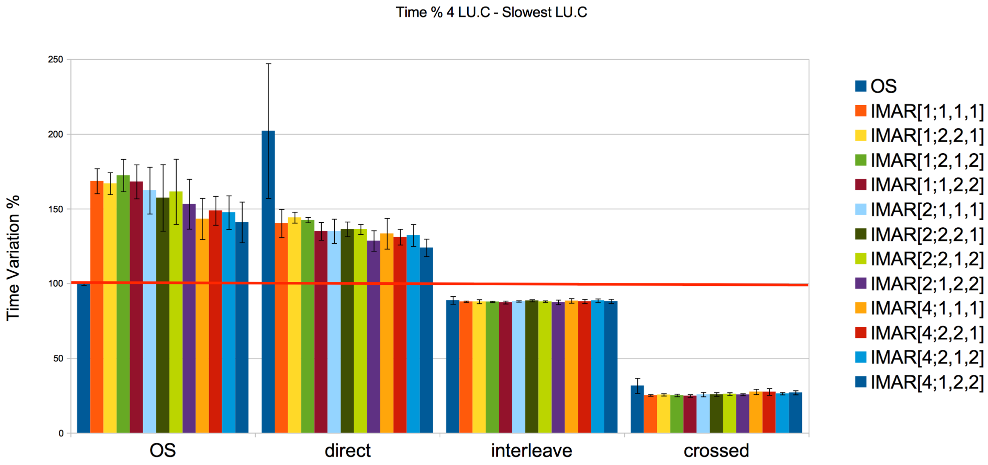


Figure 4.23: Variations of the slowest lu.C for **4 lu.C** with respect to the baseline.

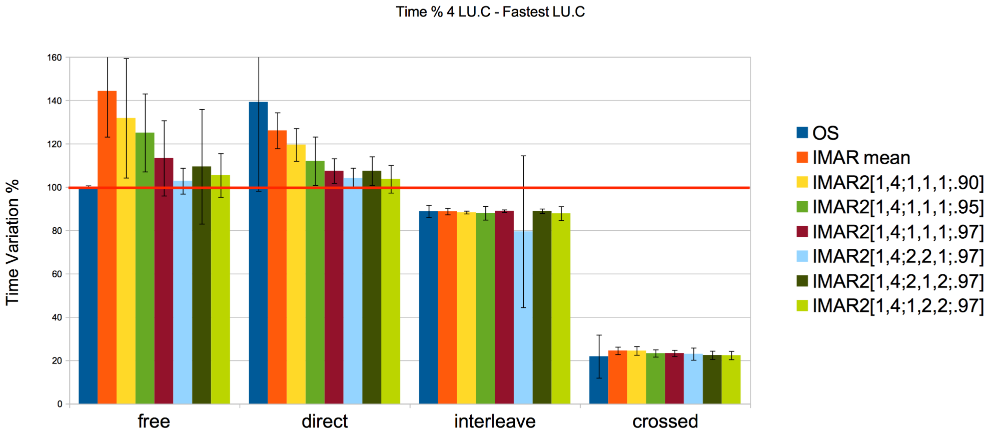


Figure 4.24: Variations of the fastest lu.C for **4 lu.C** with respect to the baseline, with IMAR^2.

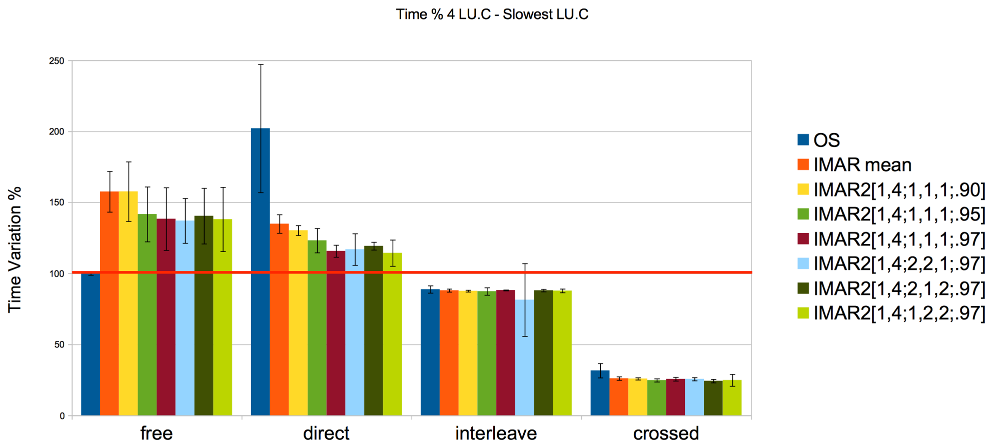


Figure 4.25: Variations of the slowest lu.C for **4 lu.C** with respect to the baseline, with IMAR^2.

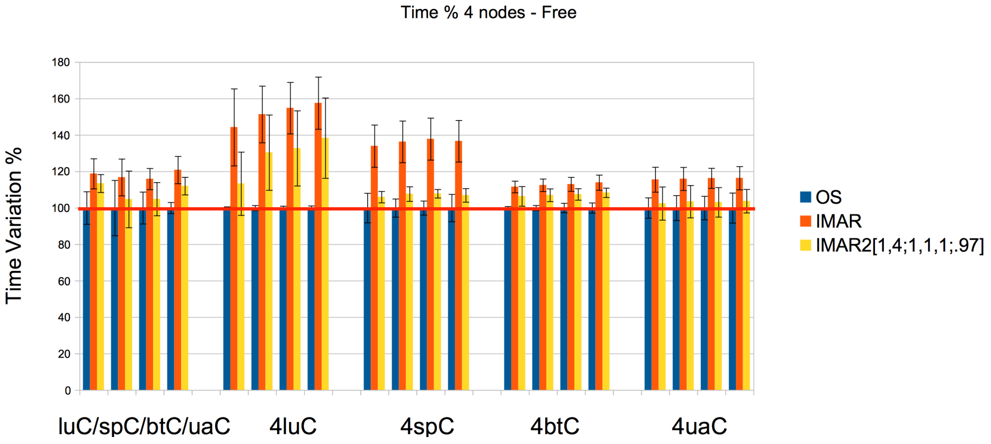


Figure 4.26: Mean results for free test with 4 nodes.

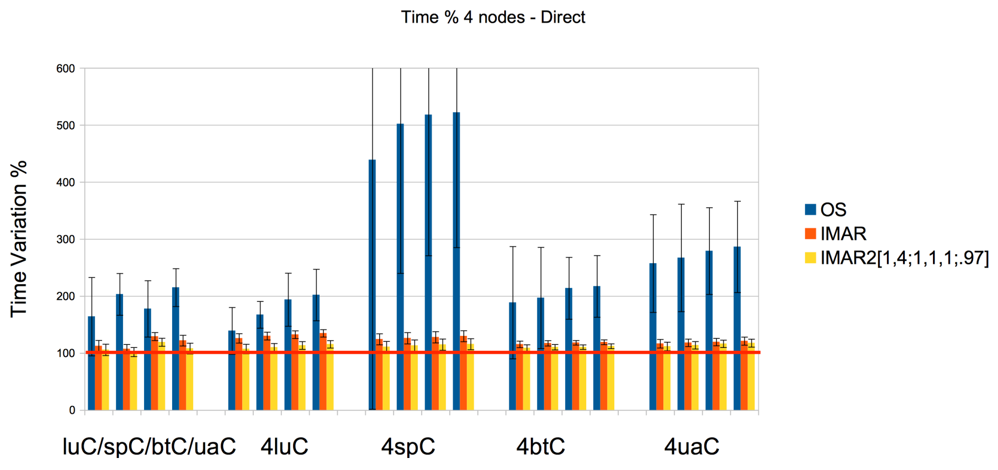


Figure 4.27: Mean results for direct test with 4 nodes.

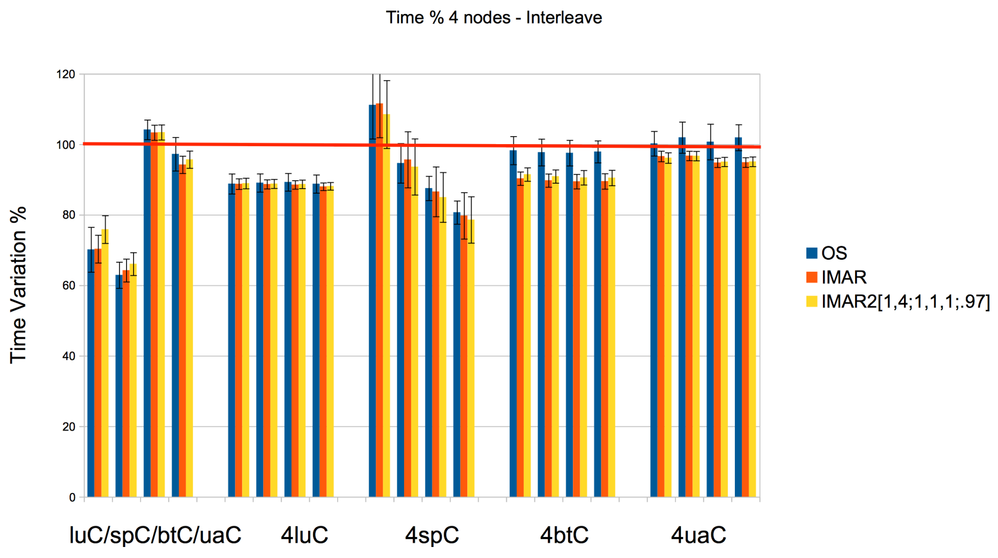


Figure 4.28: Mean results for interleave test with 4 nodes.

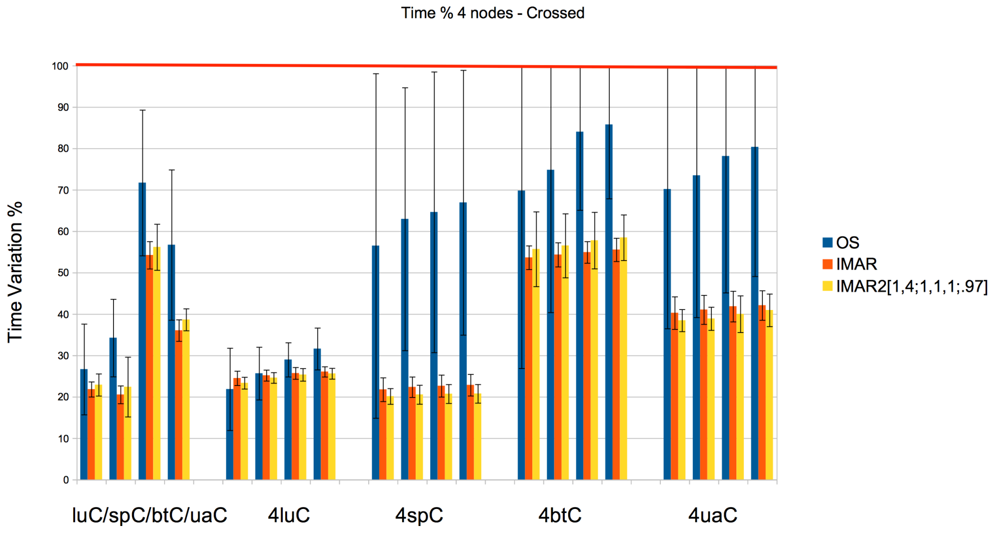


Figure 4.29: Mean results for crossed test with 4 nodes.

4.6 Recap

Modern multicore systems present complex memory hierarchies, and make load balancing, data locality and thread affinity important issues to obtain high performance. In this chapter, thread migration algorithms, based on the optimisation of 3DyRM parameters, were used to increase performance. The proposed techniques improve execution times when thread locality is poor and the OS cannot improve thread placement during runtime. A multiobjective optimisation method, weighted product, was proposed to combine the 3DyRM parameters. In Section 4.2, three migration algorithms have been presented.

In Section 4.3, we described how, using hardware counters, the performance of each thread in the system could be obtained in runtime with low overhead, and a tool was implemented to perform thread migration and allocation during runtime, applying different migration strategies and algorithms, tuned by a set of factors.

The IMA algorithm was based on interchanging, at regular intervals, the worst performing thread with another. As case studies for this algorithm, analysis of the parallel SDOT and SAXPY routines were performed, in Section 4.4.4, in different scenarios to explore different locality and affinity properties. Also, different thread migration strategies were considered, relative to the selection of the threads to be interchanged. This way, two thread migration strategies were used to minimise the detrimental effects of the memory affinity in parallel codes on a multicore system. The results show how, given a bad distribution of threads and data, the OS by itself is not able to detect and correct it, and this greatly influences the performance. Improvements up to 25 % with respect to the OS were achieved in cases with low locality and affinity (CROSSED scenarios), whereas in the opposite situation (IDEAL scenarios) only a small loss in performance was observed in some situations.

The IMAR algorithm uses collected information about previous performance for each thread to guide thread migration decisions. This algorithm was tested on a server using benchmarks from the NPB-OMP (in Section 4.5). On complex systems, where NUMA effects are more pronounced, a poor allocation of threads and data can degrade performance by a factor of up to 5 or 6. Given a poor distribution of threads and data, the OS by itself is not able to detect and correct it, which greatly influences performance. The IMAR algorithm was able to improve execution by up to 70%. However, small performance losses were obtained in cases where the thread configuration was initially good.

The IMAR^2 algorithm can be considered a refining of the IMAR algorithm. It is based on the concept of evaluating the effects on the system total performance of previous migrations and acting accordingly. Specifically, IMAR^2 is based on IMAR, but adds rollback and changes in the period between migrations. This provides for greater tuning and performs better for those cases where migrations are unnecessary, while still improving the performance for cases with low initial performance. Generally, IMAR^2 was superior to IMAR, which, in turn, was superior to allowing the OS to self-optimise.

Manuals for the applications presented in this chapter, as well as the source code of the tools, can be found at the CiTIUS git repository [62].

Conclusions and Future Work

Nowadays, most computer systems are multicore and even multiprocessor based. In these systems, the behaviour of memory accesses from each thread to the different memory nodes is one of the most significant aspects influencing the performance of any code. This fact is more and more relevant as the memory wall increases.

In this work, this issue has been tackled from two points of view. On one hand, from the point of view of a programmer of parallel applications, tools and models to characterise the behaviour of codes and help her in their implementation have been developed. In Chapter 2 a set of tools to gather and show information about memory accesses was presented. In Chapter 3 a performance model for parallel applications, the 3DyRM, along with tools to help to characterise performance, were presented. On the other hand, from the point of view of a user of parallel applications, a migration tool, based on the 3DyRM, to automatically select and adapt, during runtime, the thread placement on a system to improve their performance, has been developed. This tool and its operation, was described in Chapter 4. All these tools make the use of runtime performance data obtained from Hardware Counters (HC) present on Intel processors. Precise Event Based Sampling (PEBS) on modern Intel processors and Event Address Counters (EARs) on the legacy Itanium 2 processors were used.

Analysis of memory accesses in SMPs

HC give detailed information about the memory addresses accessed by a code and the latency of the data loads. In Chapter 2, it was shown that, although only a sampled trace can be obtained with these counters, it gives valuable information about locality, memory access patterns and affinity among data and cores. This means that useful information can be gathered with a low overhead and a minimum impact on the application behaviour. The set of tools developed in this work simplify the process of obtaining and studying hardware counter data. Their focus on memory accesses makes them more suitable for analysing the memory behaviour than other general purpose ones. The tools are adaptable for their use in any environment, architecture or language. First, because they are split between a visualisation tool and a data capture tool, both independent. Second, because the data capture tool has been designed to be easily modifiable. The data visualisation tool can be used to do an statistical study of the captured events. The information and graphs provided by the tool can be used to analyse the influence of false sharing, coherence implications, and other memory related issues.

To test these tools, parallel OpenMP shared memory programs were considered. The SpMxV problem was used as a case study in an Itanium 2 based system. This study showed how the tools can be used to analyse the behaviour of the memory hierarchy in a friendly way. Studying the SpMxV problem as an example, it was found that the information obtained is useful to model the execution of a parallel program by analysing its memory access patterns and latencies. For example, it is easy to identify which threads have a greater workload, relative to their input data, since it determines the number of memory accesses, their locality, and latency. Programmers may find this information useful to increase the performance of their applications. A regular parallel code, SDOT, was also used as a different case study. This study focused on the detection of false sharing on a Xeon based system. Both examples show the usefulness of these tools for analysing the effects of the cache coherence mechanisms on data partitioning in parallel environments.

Performance models based on runtime information

Berkeley Roofline Model is a useful and simple model to characterise performance. It ties together floating-point performance, operational intensity, and memory performance in a 2D graph. Nevertheless, it hides some important characteristics, specially in multicore and many-core, NUMA or heterogeneous systems. A set of extensions to the Berkeley Roofline Model have been presented in Chapter 3. These extensions aim to give more expressivity to the model while retaining its simplicity as much as possible. The first extension, the Dynamic Roofline Model (DyRM) shows the evolution of an application in runtime. This highlights differences during execution, such as different phases in the execution. This model is defined for each core in a multicore or multiprocessor system, since the behaviour of an application do not need to be the same on every core. The second extension builds on the DyRM, it is the Latency Extended Dynamic Roofline Model (3DyRM). The 3DyRM adds a third dimension to the graph: the memory latency.

To simplify the obtention of the models a set of tools were implemented. A data capture tool, based on the one developed for memory accesses, uses PEBS to obtain performance data during execution with a low overhead. Due to the constrains of the HC and the PEBS when measuring floating point operations, two new models, the iDyRM and the i3DyRM, were defined. These models use instructions instead of floating point operations for the performance metrics. A performance data visualisation tool helps create the DyRM and 3DyRM models (and the iDyRM and i3DyRM), alongside other performance graphs and statistics.

A set of cases on two different multicore systems were used to show their usefulness. These cases are instances of the NPB-OMP benchmarks. Using the models, it can be shown how these benchmarks present complex behaviours and imbalances on multicore systems. These problems can be modelled with these tools and the DyRM and 3DyRM models. Using HC, influence in the normal execution of the applications is minimal, and a realistic model of their performance can be made. Thanks to the DyRM, a program's behaviour, its phases or imbalances, can be more easily detected, making it easier to correct performance issues. It has been shown how some applications may have different phases that require different optimisations approaches, which are detected by the 3DyRM and may be obscured in a regular Roofline Model. It has been proved how the latency axis of the 3DyRM can help to detect memory issues that are not apparent in the operational intensity of the regular Roofline Model.

Thread migration based on runtime information

The complex memory hierarchies present in modern multicore and NUMA systems make load balance, data locality and thread affinity to be important issues for obtaining good performance. In Chapter 4, the effect of thread and data placement on performance was tested in a series of benchmarks. In some cases, a bad thread and data placement resulted in a performance more than 6 times worse. To alleviate the detrimental effects of a random placement and to improve the performance of parallel codes and concurrent applications a thread migration tool has been presented. Using different thread migration algorithms, based in the optimisation of the parameters defined in the 3DyRM and the i3DyRM, the performance of a system was measured and improved. By using HC, the performance of each thread in the system can be obtained in runtime with low overhead. These algorithms use variations of a multiobjective optimisation method, in particular, a weighted product method. Three different migration algorithms were tested in a variety of scenarios.

A first approach, the IMA algorithm, was based in interchanging, at regular time intervals, the worst performing thread with another one. This other thread could be either one of the best performing or one of the worst performing, depending on the strategy. This algorithm was tested on parallel SDOT and SAXPY routines, modified to be executed in different scenarios, to explore different locality and affinity properties. Both migration strategies were used to minimise the detrimental effects of the memory affinity in these codes, when executed in a dual processor system. The results showed how, given a bad distribution of threads and data, the OS by itself is not able to detect and correct it, and this greatly influences the performance. Improvements up to 25% with respect to the OS migration were achieved in cases with low locality and affinity. In situations were locality and affinity were already good from the beginning, a limited loss of performance was observed.

A second approach, the IMAR algorithm, was designed to operate in more complex systems. While still executing during runtime, it uses information about past performance for each thread to better guide the thread migration. This algorithm was tested with custom benchmarks based on the NPB-OMP on a quad processor system, were NUMA effects are more pronounced. This algorithm was able to improve the performance of the worst cases up to 70%. Nevertheless, a loss of performance was observed in cases were the thread configuration was good from the beginning. This issue leads to the IMAR^2 algorithm, a refining of the IMAR algorithm. The IMAR^2 algorithm allows for greater tuning and can detect when migrations are detrimental to the performance and roll them back. This means it performs better in those cases were migrations are mainly unnecessary, while still improving the performance in the bad cases.

In conclusion, it has been shown that managing thread migration and placement may lead to a performance improvement. It has been shown how the 3DyRM and i3DyRM are useful models not only for programmers and designers, but can be used to model the performance of a system in runtime. This way they can be used to improve the execution time and concurrency of parallel codes in NUMA systems.

Future work

As future work several lines of research remain open. Given the detailed information about memory accesses available from PEBS, and already processed by the migration tool, a next step would be to perform data migration to complement thread migration. This could be done by implementing page migration using information like the data source of load operations. This would give raise to new migration algorithms which could include new features for thread migration. Also thread and data migration on manycore processors should be studied. As a parallel improvement goal, energy efficiency could be considered. A similar approach as the one taken for performance could work with energy efficiency if it were combined with other sensors, like those of temperature and power consumption, present on processors and processor boards.

4.7 Publications

- Oscar G. Lorenzo, Juan A. Lorenzo, Dora B. Heras, Juan C. Pichel, Francisco F. Rivera, "Herramientas para la monitorización de los accesos a memoria de códigos paralelos mediante contadores hardware", Actas XXII Jornadas de Paralelismo (JP2011), La Laguna 2011, pages 651–656.

- Oscar G. Lorenzo, Juan A. Lorenzo, José C. Cabaleiro, Dora B. Heras, Marcos Suarez, Juan C. Pichel "A Study of Memory Access Patterns in Irregular Parallel Codes Using Hardware Counter-Based Tools" 2011 International Conference on Parallel and Distributed Processing Techniques and Applications. Las Vegas (USA). 2011

- Oscar G. Lorenzo, Tomás F. Pena, José C. Cabaleiro, Juan C. Pichel, Juan A. Lorenzo, Francisco F. Rivera, "Hardware Counters Based Analysis of Memory Accesses in SMPs", 10th IEEE International Symposium on Parallel and Distributed Processing with Applications, pp. 595-602. Leganés (Spain). 2012

- Oscar G. Lorenzo , Tomás F. Pena, José C. Cabaleiro, Juan C. Pichel and Francisco F. Rivera, "DyRM: A Dynamic Roofline Model Based on Runtime Information", 2013 International Conference on Computational and Mathematical Methods in Science and Engineering, pp. 965-967. Almería (Spain). 2013

- Oscar G. Lorenzo , Tomás F. Pena, José C. Cabaleiro, Juan C. Pichel and Francisco F. Rivera, "Extensión del modelo Roofline y herramientas para su uso", XXIV Jornadas de Paralelismo, pp. 157–162. Madrid (Spain). 2013

- Oscar G. Lorenzo , Tomás F. Pena, José C. Cabaleiro, Juan C. Pichel and Francisco F. Rivera, "Multiobjective optimization technique based on monitoring information to increase the performance of thread migration on multicores", IEEE International Conference on Cluster Computing, pp. 416-423. Madrid (Spain). 2014

- Oscar G. Lorenzo, Tomás F. Pena, José C. Cabaleiro, Juan C. Pichel, Juan A. Lorenzo, Francisco F. Rivera, "A hardware counters based toolkit for the analysis of memory accesses in SMPs", Concurrency and Computation: Practice and Experience, vol. 26, no. 6, pp. 1328-1341. 2014

- Oscar G. Lorenzo, Tomás F. Pena, José C. Cabaleiro, Juan C. Pichel and Francisco F. Rivera, "Using an extended Roofline Model to understand data and thread affinities on NUMA systems", Annals of Multicore and GPU Programming, vol. 1, no. 1, pp. 56-67. 2014

- Oscar G. Lorenzo, Tomás F. Pena, José C. Cabaleiro, Juan C. Pichel and Francisco F. Rivera, "3DyRM: a dynamic roofline model including memory latency information", Journal of Supercomputing, vol. 67, pp. 696-708. 2014

- Oscar G. Lorenzo , Tomás F. Pena, José C. Cabaleiro, Juan C. Pichel and Francisco F. Rivera, "Study of data locality and thread affinity on multicore systems using the Roofline Model", I Jornadas de Programación Paralela Multicore y GPU, pp. 67–76. Granada (Spain). 2014

- Oscar G. Lorenzo , Tomás F. Pena, José C. Cabaleiro, Francisco F. Rivera and Dimitrios S. Nikolopoulos, "Power and Energy Implications of the Number of Threads Used on the Intel Xeon Phi", II Jornadas de Programación Paralela Multicore y GPU, pp. 1–8. Cáceres (Spain)

Resumo da Tese

Hoxe en día, a maioría dos sistemas de computación son multicore e mesmo multiprocessador. Nestes sistemas, o comportamento dos accesos á memoria de cada fío para os distintos nodos de memoria é un dos aspectos que máis significativamente afectan o rendemento de calquera código. Este feito é cada vez máis relevante a medida que aumenta o chamado "memory wall".

Neste traballo, esta cuestión foi abordada baixo dous puntos de vista. Desde o punto de vista dun programador de aplicacións paralelas, desenvolvéronse ferramentas e modelos para caracterizar o comportamento de códigos e axudao para a súa aplicación. Desde o punto de vista dun usuario de aplicacións paralelas, desenvolveuse unha ferramenta de migración para seleccionar e adaptar, automaticamente durante a execución, a colocación de fíos no sistema para mellorar o seu funcionamento. Todas estas ferramentas fan uso de datos de rendemento en tempo de execución obtidos a partir de Contadores Hardware (HC) presentes nos procesadores Intel.

En comparación cos "software profilers", os HC proporcionan, cunha baixa sobrecarga, unha información de rendemento detallada e rica referente ás unidades funcionais, caches, acceso á memoria principal por parte da CPU, etc. Outra vantaxe de usalos é que non precisa ningunha modificación do código fonte. Con todo, os tipos e os significados dos contadores hardware varían dunha arquitectura a outra debido á variación nas organizacións do hardware. Ademais, pode haber dificultades para correlacionar as métricas de rendemento de baixo nivel co código fonte orixinal. O número limitado de rexistros para almacenar os contadores moitas veces pode forzar aos usuarios a realizar múltiples medicións para recoller todas as métricas de rendemento desexadas.

En concreto, neste traballo, utilizáronse os Precise Event Based Sampling (PEBS, MOSTRAXE BASEADO EN EVENTOS PRECISOS) nos procesadores Intel modernos e os Event Address Register (EARs, REXISTROS DE ENDEREZO DE EVENTO) nos procesadores Itanium 2.

O procesador Itanium 2 ofrece un conxunto de rexistros, os EARs que rexistran os enderezos de instrución e datos dos fallos caché, e os enderezos de instrución e datos de fallos de TLB [25]. Cando se usan para capturar fallos caché, os EARs permiten a detección das latencias maiores de 4 ciclos. Xa que os accesos de punto flotante sempre provocan un fallo (os datos de punto flotante son sempre almacenados na L2D), calquer acceso pode ser potencialmente detectado. Os EARs permiten a mostraxe estatística, configurando un contador de rendemento para contar as aparicións dun determinado evento.

O PEBS usa un mecanismo de interrupción cos HC para almacenar un conxunto de información sobre o estado da arquitectura para o procesador. A información ofrece o estado arquitectónico da instrución executada despois da instrución que causou o evento. Xunto con esta información, que inclúe o estado de todos os rexistros, os procesadores Sandy Bridge posúen un sistema de medición da latencia a memoria. Ista é un medio para caracterizar a latencia de carga media para os diferentes niveis da xerarquía de memoria. A latencia é medida dende a expedición da instrucción ata cando os datos son globalmente observables, e dicir, cando chegan ao procesador. Ademáis da latencia, o PEBS permite coñecer a orixe dos datos e o nivel de memoria de onde se leron. A diferenza dos EARs, o PEBS permite tamén medir a latencia de operacións enteiras ou de almacenamento de datos.

Análise de accesos a memoria en SMPs

Os HC dan información detallada sobre os enderezos de memoria accedidos por un código e a latencia das cargas de datos. Aínda que só unha traza de mostras pode ser obtida con estes contadores, ista dá información valiosa sobre localidade, patróns de acceso de memoria e afinidade entre os datos e núcleos. Isto significa que se pode recoller información útil cunha baixa sobrecarga e un impacto mínimo sobre o comportamento da aplicación.

O proceso de recollida de información sobre os accesos á memoria dos contadores EAR ou PEBS non é tan sinxelo como interactuar con outros tipos de HC. No canto de ter que realizar só a lectura dun valor desde simples contadores, hai a necesidade de procesar un "buffer"de datos a intervalos irregulares durante a execución dun programa. Debido a isto, desenvolveuse un conxunto de ferramentas para simplificar o proceso de obtención e estudo dos datos dos contadores hardware. O seu foco en accesos á memoria tórnaos máis axeitado para a análise do comportamento da memoria que outras ferramentas de uso xeral. As ferramentas desenvolvidas son adaptables para o seu uso en calquera tipo de entornos, arquitecturas ou linguaxes de programación. En primeiro lugar, porque están divididos entre unha ferramenta de visualización e unha ferramenta de captura de datos, ambas independentes entre si. En segundo lugar, porque a ferramenta de captura de datos está deseñada para ser facilmente modificable.

A ferramenta de captura de datos é unha ferramenta de liña de comandos que pode traballar xunto a calquera programa xa compilado e que reúne a información de acceso de memoria durante a súa execución. A ferramenta acepta tres opcións: o evento para supervisar (fallos cache ou TLB, en caso de EARs, instrucións de carga ou almacenamento, en caso de PEBS), o período de mostraxe e a latencia de carga mínima a partir da cal os eventos son capturados. De ser necesario un seguimento máis detallado só dalgunhas partes dun código ou dun intervalo de datos limitado, e se o código fonte a supervisar está dispoñible, a ferramenta de instrumentación pode ser usada. Esta ferramenta axuda a engadir directamente o código para EAR ou PEBS a un código fonte para que o programa se poida supervisar. O usuario só precisa indicar, mediante directivas no código fonte, onde comezar e rematar a captura de datos.

A ferramenta de visualización de datos pode usarse para facer un estudo estatístico dos eventos capturados. Permite clasificar eventos capturados en categorías segundo o seu enderezo de memoria ou o fío. Estes eventos poden ser tanto fallos de caché coma fallos de TLB. Amosa os eventos capturados nun histograma, que se pode delimitar polos enderezos inicial e final do rango de memoria virtual estudado, ou cunha categoría para cada fío. A información e gráficos proporcionados pola ferramenta poden ser utilizados para analizar a influencia do "false sharing", as implicacións de coherencia e outros problemas relacionados coa memoria.

Para probar estas ferramentas, utilizáronse programas paralelos de memoria compartida

OpenMP. O problema SpMxV usouse como caso de estudo nun sistema baseado en Itanium2. Este estudo demostrou que as ferramentas poden usarse para analizar o comportamento da xerarquía de memoria. Estudando o problema SpMxV como exemplo, verificouse que a información obtida é útil para modelar a execución dun programa paralelo, analizando os seus patróns de acceso á memoria e as latencias. Por exemplo, foi doado ver qué fíos teñen unha maior carga de traballo, respecto dos seus datos de entrada, xa que determina o número de accesos á memoria, a súa localidade e latencia. Os programadores poden atopar esta información útil para aumentar o rendemento das súas aplicacións. Un código paralelo regular, SDOT, usouse como un caso de estudo diferente. Este estudo incidiu sobre a detección de "false sharing"nun sistema baseado en Xeon. Ambos exemplos demostran a utilidade destas ferramentas para analizar os efectos dos mecanismos de coherencia caché no particionamento de datos en entornos paralelos.

Modelos de rendemento baseados en información en tempo real

O Berkeley Roofline Model (RM) [85] é un modelo útil e sinxelo para caracterizar o rendemento. É un modelo sinxelo que ofrece directrices de actuación e información sobre o comportamento dun programa, información que pode axudar aos programadores a entender o desempeño ou rendemento dos seus códigos. Nembargantes, é probable que no futuro próximo o ancho de banda fóra do chip de memoria sexa o recurso limitante na maioria das situacións. Así, un modelo que relaciona o desempeño do procesador e o tráfico entre o chip e a memoria é necesario. Para este obxectivo, a intensidade operacional (OI) úsase para significar número de operacións en punto flotante (Flops) por byte de tráfico DRAM. O total de bytes accedidos defínense como aqueles que se dirixen á memoria principal despois de ser filtradas pola xerarquía da caché. É dicir, o tráfico mídese entre as cachés e a memoria en vez de entre o procesador e as caches. Así, a intensidade operacional suxire o ancho de banda DRAM necesaria por un núcleo computacional nun ordenador particular. O RM une o rendemento de punto flotante, a intensidade operacional e o rendemento da memoria nun único gráfico 2D. Sen embargo, esconde algunhas características importantes, especialmente en sistemas multicore e manycore, NUMA (Non Uniform Memory Access, ACCESO A MEMORIA NON UNIFORME) ou heteroxéneos.

Nesta tese propóñense un conxunto de extensións para o RM. Estas extensións pretenden dar máis expresividade ao modelo, mantendo a súa sinxeleza, na medida do posible. A primeira extensión, o Dynamic Roofline Model (DyRM, MODELO RM DINÁMICO) mostra a evolución dunha aplicación durante a súa execución. Isto permite destacar diferenzas durante a execución, como as distintas fases na execución. O DyRM é, esencialmente, equivalente a dividir o tempo de execución dun código en partes, e crear un RM para cada unha, para, a continuación, combinalas nun único gráfico. Este modelo define un comportamento para cada núcleo nun sistema multiprocesador ou multinúcleo, xa que o comportamento dunha aplicación non ten por qué ser o mesmo en cada núcleo. No DyRM, úsanse eixes lineais en vez dos eixos logarítmicos do RM orixinal para amosar mellor pequenas diferenzas no comportamento.

A segunda extensión baséase no DyRM e é o Latency Extended Dynamic Roofline Model (3DyRM, DyRM extendido en latencia). A OI, para caracterizar o rendemento, pode ser insuficiente, especialmente en sistemas NUMA. O RM define límites superiores de rendemento, pero nun sistema NUMA, a distancia e a conexión con celdas de memoria de diferentes núcleos pode implicar variacións na latencia de acceso. Variacións no tempo de acceso poden causar valores diferentes nos GFLOPS (Giga Floating Point Operations per Second, UN MILLÓN DE OPERACIÓNS EN PUNTO FLOTANTE POR SEGUNDO) para cada núcleo, aínda que cada núcleo execute o mesmo número de operacións. Nestes casos, a OI pode permanecer prácticamente constante, agochando o feito de que o baixo rendemento é debido ao subsistema de memoria. Por iso o 3DyRM engade unha terceira dimensión ao gráfico: a latencia de memoria.

Para simplificar a obtención do modelo, nesta tese desenvolvéronse un conxunto de ferramentas. Unha ferramenta de captura de datos, sobre a base da que foi deseñada para os accesos á memoria, utiliza PEBS para obter os datos de rendemento durante a execución, cunha baixa sobrecarga. Debido as restriccións dos HC e os PEBS cando miden operacións de punto flotante, definíronse dous novos modelos, o iDyRM eo i3DyRM. Estes modelos utilizan instrucións no canto de operacións de punto flotante para as métricas de rendemento. Unha ferramenta de visualización de datos de rendemento axuda a crear os modelos DyRM e 3DyRM (máis os iDyRM e i3DyRM), xunto a outros gráficos de rendemento e estatísticas.

Para mostrar a utilidade destas ferramentas, utilizáronse un conxunto de casos de estudo en dous sistemas multicore diferentes. Estes casos utilizan instancias dos programas de proba NPB-OMP. Usando os modelos pode mostrase como estes "benchmarks"presentan comportamentos complexos e desequilibrios nos sistemas multicore. Estes problemas poden ser facilmente modelados con estas ferramentas e os modelos DyRM e 3DyRM. Usando HC, a influencia na execución normal das aplicacións é mínimo, e se pode obter un modelo realista do seu rendemento. Grazas ao DyRM, o comportamento dun programa, as súas fases ou desequilibrios poden detectarse máis facilmente, o que o fai máis fácil resolver problemas de rendemento. Demostrouse como algunhas aplicacións poden ter distintas fases que requiren diferentes enfoques de optimizacións, as cales son detectadas polo 3DyRM e poden ser obscurecidos nun RM estándar. Demostrouse que o eixe de latencia do 3DyRM pode axudar a detectar problemas de memoria que non son evidentes na intensidade operacional do RM regular.

Migración de fíos baseada en informanción en tempo de execución

As complexas xerarquías de memoria presentes nos sistemas multicore e NUMA modernos fan do balanceo da carga, da localidade de datos e da afinidade de fíos cuestións importantes para a obtención dun bo rendemento. Nesta tese, comprobouse o efecto da colocación de fíos e de datos sobre o rendemento usando unha serie de "benchmarks". Nalgúns casos unha mala colocación de fíos e datos deu lugar a un rendemento máis de 6 veces peor. Para paliar os efectos prexudiciais dunha colocación aleatoria e para mellorar o rendemento, implementouse unha ferramenta de migración dinámica de fíos. Esta ferramenta encárgase de monitorizar todos os fíos do sistema (usando HC) e decide a súa colocación utilizando un algoritmo de migración. Ao usar HC, o desempeño de cada fío do sistema pode obterse en tempo de execución con baixo custo. Usando diferentes algoritmos de migración de fíos, con base na optimización dos parámetros definidos no 3DyRM e i3DyRM, o rendemento dun sistema mediuse e mellorouse. Estes algoritmos utilizan variacións dun método de optimización multiobxectivo, en particular, o método do produto ponderado. Todos combinan os tres parámetros do 3DyRM nunha soa medida de rendemento coa expresión:

$$P_{ij} = \frac{\text{GFLOPS}_{ij}^{\beta} \cdot \text{flopsB}_{ij}^{\gamma}}{\text{latency}_{ij}^{\alpha}} \qquad (4.6)$$

$$i = 1, \ldots, M \qquad j = 0, \ldots, N-1$$

sendo P_{ij} o rendemento para o i-ésimo fío, M o número de fíos, no j-ésimo nodo e N o número de nodos do sistema. $\text{GFLOPS}_{ij}^{\beta}$ é a medida de GFLOPS do i-ésimo fío executado no j-ésimo nodo no momento de calcular P_{ij}, escalado por certo valor β. A súa contribución a P_{ij} pódese modificar cambiando β. O mesmo ocorre para $\text{flopsB}_{ij}^{\gamma}$ e $\text{latency}_{ij}^{\alpha}$ con γ e α, respectivamente. Tres algoritmos de migración diferentes foron probados nunha variedade de escenarios.

A primeira versión, o algoritmo IMA, baseouse no intercambio regular do fío con peor rendemento con outro. Este outro fío podería ser calquera, ben un dos de mellor desenpeño ou un dos de peor desempeño, dependendo da estratexia. Esta proposta probouse en paralelo coas rutinas SDOT e SAXPY, modificadas para seren executadas en diferentes escenarios, para poder así explorar diferentes propiedades de localidade e afinidade. Ambas estratexias de migración foron usadas para minimizar os efectos prexudiciais da afinidade de memoria nestes códigos, cando se executan nun sistema de dous procesadores. Os resultados mostraron como, dada unha mala distribución de fíos e de datos, o sistema operativo por si só non é capaz de detectalos e corrixilos, o que inflúe grandemente no rendemento. Melloras de ata o 25 % en relación ao SO foron alcanzadas nos casos de baixa localidade e afinidade. En situacións onde a localidade e a afinidade xa era boa dende o principio, observouse unha pequena perda de rendemento.

Unha segunda versión, o algoritmo IMAR, deseñouse para operar en sistemas máis complexos. Durante o tempo de execución, o IMAR usa información do rendemento pasado de cada fío para orientar mellor a migración de fíos. A ferramenta de migración garda para cada fío o seu rendemento (os P_{ij}) nos diferentes nodos dun sistema NUMA. En cada iteración o algoritmo detecta o fío que peor rendemento ten respecto ao seu proceso. Despois escolle a onde migralo dependendo dos valores de P_{ij} anteriores. Se xa existen fíos ocupando o lugar de destino, os valores P_{ij} pasados deses fíos son tidos en conta para atopar a migración máis axeitada para ambos. Xa que non toda a información pode estar dispoñible, e pensando que o comportamento do sistema pode variar, valoráronse distintas opcións de migración, escollendose como mellor solución unha baseada nunha "lotería poderada".

Este algoritmo foi probado con tests personalizados baseados nos NPB-OMP nun sistema de catro procesadores, onde os efectos NUMA son máis pronunciados. Estes tests executan catro procesos dos "benchmarks"NPB-OMP á vez, fixando os núcleos que usarán ao principio e os nodos nos que gardarán os datos, para poder forzar diferentes situacións. O IMAR foi capaz de mellorar ata un 70 % o rendemento dos peores casos, aqueles onde os procesos son executados coa memoria en nodos diferentes aos que executan os fíos. Con todo, observouse unha perda de rendemento nos casos nos que a configuración de fíos era boa desde o principio, e dicir, aquela onde os datos e os fíos residen no mesmo nodo. Esta cuestión levou a propoñer o algoritmo IMAR^2, como un refinamento do algoritmo IMAR.

O algoritmo IMAR^2 permite unha maior regulación e pode detectar cando as migracións son prexudiciais para o rendemento e revertelas. Tamén permite modificar automáticamente a frecuencia das migracións, dependendo do resultado no rendemento total das mesmas. Isto significa que ten mellor comportamento nos casos onde as migracións son innecesarias, mentres que segue mellorando o rendemento nos casos malos. En conclusión, demostrouse que a xestión da migración e o posicionamento de fíos pode conducir a unha mellora do rendemento. Demostrouse tamén que os 3DyRM e i3DyRM son uns modelos útiles non só para os desenvolvedores e deseñadores, senón tamén para modelar o rendemento dun sistema en tempo de execución. Deste xeito se pode empregar para mellorar o tempo de execución e a concurrencia en sistemas NUMA.

Traballo futuro

Como traballo futuro, varias liñas de investigación permanecen abertas. Dada a información detallada sobre accesos á memoria dispoñible usando os EAR e os PEBS, e xa procesados pola ferramenta de migración, o seguinte paso sería realizar a migración de datos para complementar a migración de fíos. Isto podería ser feito a través do uso da migración de páxinas, usando información como o enderezo dos operandos das operacións de carga de datos. Iso daría pé a novos algoritmos de migración que poden incluír novas características na migración de fíos. Tamén a migración de fíos e datos nos procesadores manycore pode ser estudada en conxunto. Como obxectivo, poderíase pensar en mellorar a eficiencia enerxética. Unha visión semellante á tomada para o rendemento podería tamén funcionar coa eficiencia enerxética, utilizando outros sensores, como os de temperatura e consumo de enerxía, presentes en procesadores e placas madre.

Appendix - NAS Parallel Benchmark Suite

The NAS Parallel Benchmarks (NPB) are a set of programs designed to evaluate the performance of parallel supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications, and consist of five kernels and three pseudo-applications in the original “pencil-and-paper” specification [3]. The benchmark suite has been extended to include new benchmarks for unstructured adaptive meshes, parallel I/O, multi-zone applications, and computational grids. Problem sizes in NPB are predefined and indicated as a number of different classes. Reference implementations of NPB are available in commonly-used programming models like MPI and OpenMP (NPB 2 and NPB 3) [30].

- The kernels are:
  - EP: An “embarrassingly parallel” kernel. It generates pairs of Gaussian random deviates according to a specific scheme. The goal of this kernel is to establish the reference point for peak performance of a given platform.
  - MG: A simplified multigrid kernel. It uses a V-cycle multigrid method to compute the solution of the 3-D scalar Poisson equation. The algorithm works continuously on a set of grids that are between coarse and fine. It tests both short and long distance data movements.
  - CG: A conjugate gradient method. It computes an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. This kernel is

used on unstructured grid computations in which it tests irregular long distance communications employing unstructured matrix vector multiplications.

  - FT: A 3-D partial differential equation solution using FFTs. This kernel performs the basis of many "spectral" codes. It is a rigorous test of long distance communications performance. FT runs three one-dimensional (I-D) FFTs, one for each dimension.
  - IS: A large integer sort. This kernel performs a sorting operation that is important, for example, in "particle method" codes. It tests both integer computation speed and communications performance.

- The pseudo applications are:

  - BT: Block Tri-diagonal solver. A simulated CFD application that uses an implicit algorithm to solve 3-dimensional (3-D) compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the x, y and z dimensions. The resulting systems are block-tridiagonal of 5x5 blocks and are solved sequentially along each dimension.
  - SP: Scalar Penta-diagonal solver. A simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples x, y and z dimensions. The resulting system has scalar pentadiagonal bands of linear equations that are solved sequentially along each dimension.
  - LU: Lower-Upper Gauss-Seidel solver. A simulated CFD application that uses symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretisation of the Navier-Stokes equations in 3-D by splitting it into block lower and block upper triangular systems.

The problem sizes available are:

- Class S: small, for quick test purposes.

- Class W: workstation size (a 90's workstation; now likely too small).

- Classes A, B, C: standard test problems; 4X size increase going from one class to the next.

- Classes D, E, F: large test problems; 16X size increase from each of the A, B, and C classes, respectively.

Bibliography

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[2] Juan Ángel Lorenzo del Castillo. Performance counter-based strategies to improve data locality on multiprocessor systems: Reordering and page migration techniques, PhD Dissertation. 2012.

[3] H. Bailey, D, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[4] V. Bhaskar. A closed queuing model with multiple servers for multithreaded architecture. *J. Computer Comm.*, 31:3078–3089, 2008.

[5] G. Bitzes and A. Nowak. The overhead of profiling using PMU hardware counters. *CERN openlab Report 2014*, 2014.

[6] J. A. Brown, L. Porter, and D. M. Tullsen. Fast thread migration via cacheworking set prediction. In *17th International Symposium on High Performance Computer Architecture*, 2011.

[7] Y. H. Chang and C. H. Yeh. Evaluating airline competitiveness using multiattribute decision making. *Omega*, 29(5):405–415, 2001.

[8] X. E. Chen and T. M. Aamodt. A first-order fine-grained multithreaded throughput model. In *IEEE 15th Int. Symp. High-performance computer architecture (HPCA)*, 2009.

[9] S. Cheng, C. W. Chan, and G. H. Huang. Using multiple criteria decision analysis for supporting decisions of solid waste management. *Environmental Science and Health, Part A: Toxic/Hazardous Substances and Environmental Engineering*, 37(5):975–990, 2002.

[10] A. Cheung and S. Madden. Performance profiling with EndoScope, an acquisitional software monitoring framework. *Proceedings of the VLDB Endowment*, 1(1):42–53, 2008.

[11] T. Constantinou, Y. Sazeides, P. Michaud, D. Fetis, and A. Seznec. Performance implications of single thread migration on a chip multi-core. *ACM SIGARCH Computer Architecture News*, 33(4):80–91, 2005.

[12] L. De Rose, Y. Zhang, and D. A Reed. SvPablo: A multi-language performance analysis system. In *Computer Performance Evaluation*, pages 352–355. Springer, 1998.

[13] R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the perfect benchmarks. *IEEE Trans. Parallel Distrib. Syst.*, 9(1):5–23, 1998.

[14] S. Eranian. Perfmon2: a flexible performance monitoring interface for Linux. In *Proc. of the 2006 Ottawa Linux Symposium*, pages 269–288. Citeseer, 2006.

[15] Galicia Supercomputing Center. Finisterrae Supercomputer. https://www.cesga.es/es/infraestructuras/computacion/finisterrae, 2010. [Online; accessed December 2015].

[16] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[17] GNU. gprof. https://sourceware.org/binutils/docs/gprof/, 2015. [Online; accessed December 2015].

[18] GNU. Octave. http://www.gnu.org/software/octave/, 2015. [Online; accessed December 2015].

[19] G. H. Golub and C. F. Van Loan. *Matrix computations*. The John Hopkins University Press, 1989.

[20] E. Gutiérrez, O. Plata, and E. L. Zapata. A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors. In *Proc. of the Int. Conf. on Supercomputing*, LNCS, pages 78–87. ACM SIGARCH, Springer-Verlag, May 2000.

[21] Z. Guz, E. Bolotin, I. Keidar, A. Mendelson, and U. C. Weiser. Manycore vs. manythread machines: Stay from the valley. *IEEE Computer Architecture Letter*, 8(1):25–28, 2009.

[22] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012.

[23] HP. HP Integrity rx7640 Server Quick Specs. http://www8.hp.com/h20195/v2/getpdf.aspx/c04140246.pdf?ver=25, 2009. [Online; accessed December 2015].

[24] HP. HP Caliper. www.hp.com/go/caliper, 2015. [Online; accessed December 2015].

[25] Intel. Dual-Core Update to the Intel Itanium 2 Processor Reference Manual. http://www.intel.com/content/www/us/en/processors/itanium/dual-core-update-itanium-2-processor-manual.html, 2015. [Online; accessed December 2015].

[26] Intel. Intel Ark, Intel Xeon Processor E5-2603. http://ark.intel.com/products/64592/, 2015. [Online; accessed December 2015].

[27] Intel. Intel VTune performance analyzer. https://software.intel.com/en-us/intel-vtune-amplifier-xe, 2015. [Online; accessed December 2015].

[28] Intel. Intel®64 and IA-32 Architectures Software Developer’s Manual Volume 3B: System Programming Guide, Part 2. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html, 2015. [Online; accessed December 2015].

[29] J. D. McCalpin. Preventing FP overcounts for AVX instructions on Sandy Bridge. https://software.intel.com/en-us/forums/software-tuning-performance-optimization-platform-monitoring/topic/564455, 2015. [Online; accessed December 2015].

[30] H. Jin, M. Frumkin, and J. Yan. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[31] M. Ju, H. Jung, and H. Che. A performance analysis methodology for multicore, multithreaded processors. *IEEE Tr. on Computers*, 63(2):276–289, 2014.

[32] A. Kleen. A NUMA API for Linux. *Novel Inc*, 2005.

[33] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis. Autopin–automated optimization of thread-to-core pinning on multicore systems. In *Transactions on high-performance embedded architectures and compilers III*, pages 219–235. Springer, 2011.

[34] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *31st International Symposium on Computer Architecture*, pages 64–75, 2004.

[35] J. Labarta, S. Girona, V. Pillet, T. Cortes, and L. Gregoris. Dip: A parallel program development environment. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *Euro-Par'96 Parallel Processing*, volume 1124 of *Lecture Notes in Computer Science*, pages 665–674. Springer Berlin / Heidelberg, 1996.

[36] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall Englewood Cliffs, 1984.

[37] C. Li, C. Ding, and K. Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, page 2. ACM, 2007.

[38] Y. Li, I. Pandis, R. Müller, V. Raman, and G. M. Lohman. NUMA-aware algorithms: the case of data shuffling. In *6th Biennial Conference on Innovative Data Systems Research (CIDR'13)*, 2013.

[39] F. Liu, F. Guo, Y. Solihin, S. Kim, and A. Eker. Characterizing and modeling the behavior of context switch misses. In *Intl. Conf. on Parallel Architectures and Compilation Techniques*, 2008.

[40] O. G. Lorenzo, J. A. Lorenzo, J. C. Cabaleiro, D. B. Heras, M. Suarez, and J. C. Pichel. A study of memory access patterns in irregular parallel codes using hardware counter-based tools. In *Int. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, pages 920–923, 2011.

[41] O. G. Lorenzo, J. A. Lorenzo, D. B. Heras, J. C. Pichel, and F. F. Rivera. Herramientas para la monitorización de los accesos a memoria de códigos paralelos mediante contadores hardware. *XXII Jornadas de Paralelismo (JP2011)*, 9:07–2011, 2011.

[42] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, J. A. Lorenzo, and F. F. Rivera. A hardware counter-based toolkit for the analysis of memory accesses in smps. *Concurrency and Computation: Practice and Experience*, 26(6):1328–1341, 2014.

[43] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, J. A. Lorenzo, F. F. Rivera, et al. Hardware counters based analysis of memory accesses in SMPs. In *Parallel and*

Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on, pages 595–602. IEEE, 2012.

[44] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera. DyRM: A dynamic roofline model based on runtime information. In *2013 International Conference on Computational and Mathematical Methods in Science and Engineering*, pages 965–967, 2013.

[45] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera. Extensión del modelo roofline y herramientas para su uso. In *XXIV Jornadas de Paralelismo*, pages 157–162, 2013.

[46] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera. 3DyRM: a dynamic roofline model including memory latency information. *The Journal of Supercomputing*, 70(2):696–708, 2014.

[47] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera. Multiobjective optimization technique based on monitoring information to increase the performance of thread migration on multicores. In *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pages 416–423. IEEE, 2014.

[48] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera. Study of data locality and thread affinity on multicore systems using the roofline model. In *I Jornadas de Programación Paralela Multicore y GPU*, pages 67–76, 2014.

[49] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera. Thread migration techniques based on dynamic roofline models and latency information. In *Avances en computación paralela, distribuida y de alto rendimiento, XXV Jornadas de Paralelismo*, pages 261–268, 2014.

[50] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera. Using an extended roofline model to understand data and thread affinities on NUMA systems. *Annals of Multicore and GPU Programming*, 1(1):56–67, 2014.

[51] C. K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices*, volume 40, pages 190–200. ACM, 2005.

[52] R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26:369–395, 2004.

[53] MathWorks. Matlab. http://www.mathworks.com, 2015. [Online; accessed December 2015].

[54] A. Mazouz, S. Touati, and D. Barthou. Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of SPEC OMP applications on intel architectures. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 273–279. IEEE, 2011.

[55] J.D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[56] McCalpin, J.D. STREAM benchmark. www.cs.virginia.edu/stream/ref.html, 1995. [Online; accessed December 2015].

[57] B. Mohr, A. D. Malony, H. C. Hoppe, F. Schlimbach, G. Haab, J. Hoeflinger, and S. Shah. A performance monitoring interface for OpenMP. In *Proceedings of the Fourth Workshop on OpenMP (EWOMP 2002)*, 2002.

[58] S. Moore, D. Cronk, K. London, and J. Dongarra. Review of performance analysis tools for MPI parallel programs. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 241–248. Springer, 2001.

[59] A. Morris, W. Spear, A. D. Malony, and S. Shende. Observing performance dynamics using parallel profile snapshots. In *Euro-Par 2008–Parallel Processing*, pages 162–171. Springer, 2008.

[60] D. Mosberger and S. Eranian. *IA-64 Linux Kernel: Design and Implementation.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[61] W. E. Nagel, A. Arnold, M. Weber, H. Ch. Hoppe, and K. Solchenbach. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12:69–80, 1996.

[62] O.G. Lorenzo. GIT repository for 3DyRM tools. https://gitlab.citius.usc.es/tf.pena/hc-thread-migration, 2015. [Online; accessed December 2015].

[63] O.G. Lorenzo and T.F. Pena and J.C. Cabaleiro and J.C. Pichel and F.F. Rivera. Thread migration strategies for NUMA systems. *Manuscript submitted for publication*, 2015.

[64] OpenMP Architecture Review Board. The OpenMP API specification for parallel programming. http://openmp.org, 2015. [Online; accessed December 2015].

[65] Oracle. hprof. http://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html, 2015. [Online; accessed December 2015].

[66] P. Mucci. Performance Application Programming Interface (PAPI). http://icl.cs.utk.edu/papi/, 2015. [Online; accessed December 2015].

[67] PAPI Topics. Counting Floating Point Operations on Intel Sandy Bridge and Ivy Bridge. http://icl.cs.utk.edu/projects/papi/wiki/PAPITopics:SandyFlops, 2015. [Online; accessed December 2015].

[68] Paradyn Project. Paradyn. http://www.paradyn.org/, 2015. [Online; accessed December 2015].

[69] perfmon2. Precise Event-Based Sampling (PEBS). http://perfmon2.sourceforge.net/pfmon_intel_core.html#pebs, 2015. [Online; accessed December 2015].

[70] J. C. Pichel, D. B. Heras, J. C. Cabaleiro, and F. F. Rivera. Increasing data reuse of sparse algebra codes on simultaneous multithreading architectures. *Concurrency and Computation: Practice and Experience*, 21(15):1838–1856, 2009.

[71] J. C. Pichel, J. A. Lorenzo, D. B. Heras, J. C. Cabaleiro, and T. F. Pena. Analyzing the execution of sparse matrix-vector product on the Finisterrae SMP-NUMA system. *The Journal of Supercomputing*, 58(2):195–205, 2011.

[72] J. C. Pichel, D. E. Singh, and J. Carretero. Reordering algorithms for increasing locality on multicore processors. In *Proc. of the IEEE Int. Conf. on High Performance Computing and Communications*, pages 123–130, 2008.

[73] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[74] S. Reza and G. T. Byrd. Reducing migration-induced cache misses. In *IEEE 26th International Parallel and Distributed Processing Symposium*, 2012.

[75] M. Schuchhardt, A. Das, N. Hardavellas, G. Memik, and A. Choudhary. The impact of dynamic directories on multicore interconnects. *IEEE Computer*, 46(10):32–39, 2013.

[76] M. Schulz and B. R. de Supinski. PN MPI tools: A whole lot greater than the sum of their parts. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. ACM, 2007.

[77] S. S. Shende and A. D. Malony. The Tau parallel performance system. *International Journal of High Performance Computing Applications*, 20(2):287–311, Summer 2006.

[78] K. S. Shim, M. Lis, O. Khan, and S. Devadas. Judicious thread migration when accessing distributed shared caches. In *Proceedings of the Third Workshop on Computer Architecture and Operating System Codesign (CAOS)*, 2012.

[79] F. N. Sibai. Simulation and performance analysis of multi-core thread scheduling and migration algorithms. In *2010 International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, pages 895–900. IEEE, 2010.

[80] A. G. Sodan. Message-passing and shared-data programming models: Wish vs. reality. In *Proc. IEEE Int. Symp. High Performance Computing Systems Applications*, pages 131–139, 2005.

[81] J. Torrellas, M. S. Lam, and J. L. Hennessy. False sharing and spatial locality in multiprocessor caches. *Computers, IEEE Transactions on*, 43(6):651–663, 1994.

[82] T. Walsh. *Generating miss rate curves with low overhead using existing hardware*. PhD thesis, University of Toronto, 2009.

[83] M. Wang, S. Liu, S. Wang, and K. K. Lai. A weighted product method for bidding strategies in multi-attribute auctions. *Systems Science and Complexity*, 23(1):194–208, 2010.

[84] M. V. Wilkes. The memory gap and the future of high performance memories. *ACM SIGARCH Computer Architecture News*, 29(1):2–7, 2001.

[85] S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009.

[86] Intel Developer Zone. Fluctuating FLOP count on Sandy Bridge. http://software.intel.com/en-us/forums/topic/375320, 2014. [Online; accessed September 2015].

List of Figures

List of Tables